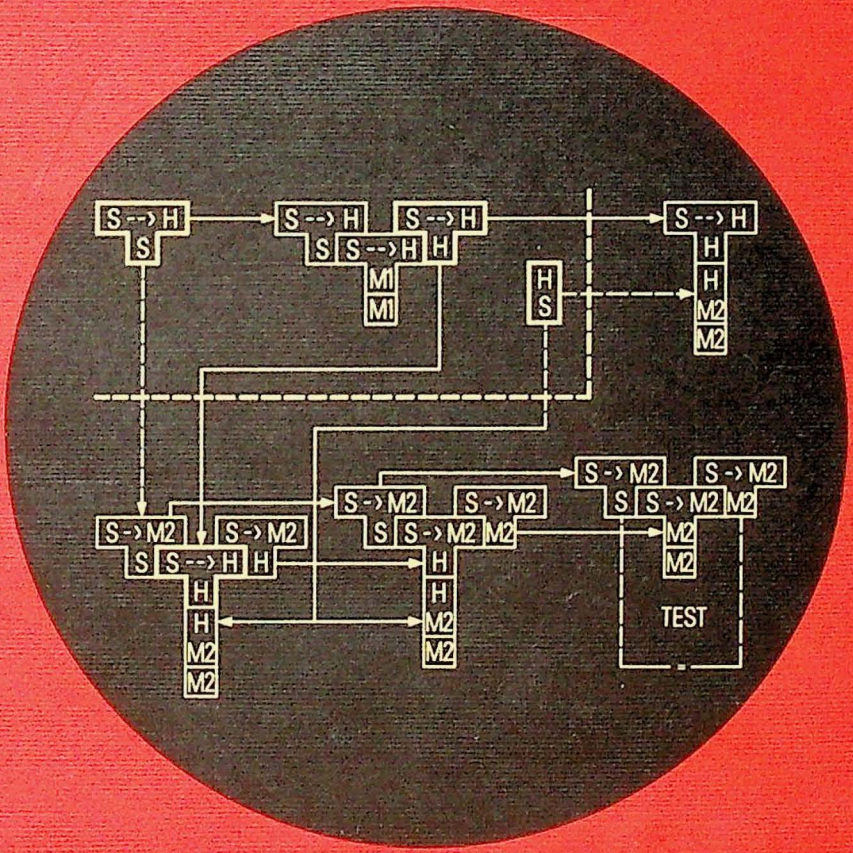
N. Wirth

# Compilerbau




Teubner Studienbücher
Informatik

# Teubner Studienbücher

## Informatik

Ehrig et al.: **Universal Theory of Automata**
A Categorical Approach
240 Seiten. DM 24,80

Giloi: **Principles of Continuous System Simulation**
Analog, Digital and Hybrid Simulation in a Computer Science Perspective
172 Seiten. DM 25,80 (LAMM)

Hotz: **Informatik: Rechenanlagen**
Struktur und Entwurf. 136 Seiten. DM 16,80 (LAMM)

Kandzia/Langmaack: **Informatik: Programmierung**
234 Seiten. DM 22,80 (LAMM)

Kupka/Wilsing: **Dialogsprachen**
168 Seiten. DM 19,80 (LAMM)

Maurer: **Datenstrukturen und Programmierverfahren**
222 Seiten. DM 26,80 (LAMM)

Oberschelp/Wille: **Mathematischer Einführungskurs für Informatiker**
Diskrete Strukturen. 236 Seiten. DM 19,80 (LAMM)

Schnorr: **Rekursive Funktionen und ihre Komplexität**
191 Seiten. DM 24,80 (LAMM)

Spaniol: **Arithmetik in Rechenanlagen**
Logik und Entwurf. 208 Seiten. DM 24,80 (LAMM)

Wirth: **Algorithmen und Datenstrukturen**
376 Seiten. DM 26,80 (LAMM)

Wirth: **Compilerbau**
Eine Einführung. 96 Seiten. DM 14,80 (LAMM)

Wirth: **Systematisches Programmieren**
Eine Einführung. 2. Aufl. 160 Seiten. DM 18,80 (LAMM)

## Mathematik

Böhmer: **Spline-Funktionen**
Theorie und Anwendungen. 340 Seiten. DM 26,80

Clegg: **Variationsrechnung**
138 Seiten. DM 16,80

Collatz: **Differentialgleichungen**
Eine Einführung unter besonderer Berücksichtigung der Anwendungen
5. Auflage. 226 Seiten. DM 22,80 (LAMM)

Collatz/Krabs: **Approximationstheorie**
Tschebyscheffsche Approximation mit Anwendungen. 208 Seiten. DM 26,80

Constantinescu: **Distributionen und ihre Anwendung in der Physik**
144 Seiten. DM 17,80

Fortsetzung auf der 3. Umschlagseite

Teubner Studienbücher Informatik

N. Wirth
Compilerbau

# Leitfäden der angewandten Mathematik und Mechanik LAMM

Unter Mitwirkung von
Prof. Dr. E. Becker, Darmstadt
Prof. Dr. G. Hotz, Saarbrücken
Prof. Dr. K. Magnus, München
Prof. Dr. E. Meister, Darmstadt
Prof. Dr. Dr. h. c. F. K. G. Odqvist, Stockholm
Prof. Dr. Dr. h. c. Dr. h. c. Dr. h. c. E. Stiefel, Zürich

herausgegeben von
Prof. Dr. Dr. h. c. H. Görtler, Freiburg

Band 36

Die Lehrbücher dieser Reihe sind einerseits allen mathematischen Theorien und Methoden von grundsätzlicher Bedeutung für die Anwendung der Mathematik gewidmet; andererseits werden auch die Anwendungsgebiete selbst behandelt. Die Bände der Reihe sollen dem Ingenieur und Naturwissenschaftler die Kenntnis der mathematischen Methoden, dem Mathematiker die Kenntnisse der Anwendungsgebiete seiner Wissenschaft zugänglich machen. Die Werke sind für die angehenden Industrie- und Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler bestimmt, darüber hinaus aber sollen sie den im praktischen Beruf Tätigen zur Fortbildung im Zuge der fortschreitenden Wissenschaft dienen.

# Compilerbau

## Eine Einführung

Von Dr. Niklaus Wirth
Professor an der Eidg. Technischen
Hochschule Zürich

Mit 20 Figuren und 20 Übungen

B. G. Teubner Stuttgart 1977

Prof. Dr. Niklaus Wirth

Geboren in Winterthur, Schweiz. Von 1954 bis 1958 Studium an der Eidg. Technischen Hochschule Zürich mit Abschluß als Dipl.-El.-Ing. Von 1959 bis 1960 Studium an der Université Laval, Quebec, Canada, und Erlangung des Grades M. Sc. Von 1960 bis 1963 Studium und anschließend Promotion an der University of California, Berkeley. Von 1963 bis 1967 Assistant Professor of Computer Science an der Stanford University. Von 1967 bis 1968 Assistenzprofessor an der Universität Zürich. Seit 1972 Professor für Informatik an der Eidg. Technischen Hochschule Zürich.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

**Wirth, Niklaus**
Compilerbau : e. Einf. – Stuttgart : Teubner, 1977.
(Leitfäden der angewandten Mathematik und Mechanik ; Bd. 36)
(Teubner Studienbücher : Informatik)
ISBN 3-519-02338-5

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, besonders die der Übersetzung, des Nachdrucks, der Bildentnahme, der Funksendung, der Wiedergabe auf photomechanischem oder ähnlichem Wege, der Speicherung und Auswertung in Datenverarbeitungsanlagen, bleiben, auch bei Verwertung von Teilen des Werkes, dem Verlag vorbehalten.

Bei gewerblichen Zwecken dienender Vervielfältigung ist an den Verlag gemäß § 54 UrhG eine Vergütung zu zahlen, deren Höhe mit dem Verlag zu vereinbaren ist.

© B. G. Teubner, Stuttgart 1977

Printed in Germany
Druck: J. Beltz, Hemsbach
Binderei: G. Gebhardt, Ansbach
Umschlaggestaltung: W. Koch, Sindelfingen

## Vorwort

Dieses Buch ist eine Einführung in die Theorie der formalen Sprachen und ihrer Satzzerlegung, sowie in die Technik der Compiler-Konstruktion. Diese Themen sind aus zweierlei Gründen relevant. Erstens erleichtert und vertieft die Kenntnis der Grundprinzipien eines Compilers das Verständnis für korrekte Verwendung von Programmiersprachen ganz allgemein. Zweitens ist die Beherrschung dieser Themen Voraussetzung für die fachgerechte Erstellung vieler Computer-Systeme, denen eine einfache Befehlssprache zugrunde liegt. Die Anzahl derartiger Anwendungen nimmt rasch zu, sei es im Gebiet der Prozesssteuerung, der Datenverarbeitung, oder der Betriebssysteme.

Bezüglich der Theorie der formalen Sprachen und der Methodik der syntaktischen Analyse beschränken wir uns auf das für den Bau eines einfachen Compilers minimal notwendige. Hingegen wird die systematische Entwicklung eines vollständigen Compilers für eine einfache Programmiersprache in ihren Einzelheiten dargelegt. Die Wahl der Quellsprache PL/0 ist ein Kompromiss zwischen einer Sprache, die allzu trivial wäre, um als lehrreiches Beispiel zu gelten, und einer Sprache, deren Mächtigkeit und Komplexität den wirklichen Kern und seine Hauptprinzipien verschleiern würde.

Für alle Programme wird die Sprache *Pascal* verwendet. Sie erlaubt eine transparente Darstellung komplexer Programme und Datenstrukturen, und sie eignet sich daher für diesen Themenkreis besonders gut.

Das Buch entstand aus einer Vorlesung an der ETH Zürich. Es ist eine stark überarbeitete Uebersetzung aus dem Englischen. Die Kapitel 12 und 13 wurden dabei neu hinzugefügt. Die Produktion des photo-reproduzierbaren Textes wurde vom Autor selbst vorgenommen. Dies war nur möglich unter Verwendung modernster Computersysteme für die Herstellung und Verarbeitung von Text und Illustrationen. Mein besonderer Dank gilt der Firma Xerox, die mir diese Werkzeuge in zuvorkommender Weise zur Verfügung gestellt hat.

Palo Alto, Dezember 1976 N. Wirth

## Inhalt


0. Einleitung 5
1. Definition und Struktur formaler Sprachen 6
2. Satzanalyse 10
3. Syntax Graphen 15
4. Aufbau eines Parsers für eine gegebene Syntax 18
5. Tabellen-gesteuerte Syntax Analyse 22
6. Die Uebersetzung von BNF-Produktionen in Parser-Tabellen 26
7. Die Programmiersprache PL/0 35
8. Ein Parser für PL/0 39
9. Die Behandlung von syntaktischen Fehlern 48
10. Ein Interpreter für PL/0 57
11. Die Erzeugung von Befehls-Code 60
12. Konzeption und Implementation einer Spracherweiterung: Co-Prozesse 71
13. Technik der Compilerentwicklung und -Uebertragung 79
14. Aufgabensammlung 85

Literaturhinweise 90

Anhang

Der ASCII Zeichensatz 91
Zweierpotenzen 92

Sachverzeichnis 93

# 0. Einleitung

Ziel dieses Buches ist die Entwicklung einer einfachen, rudimentären Programmiersprache und eines zugehörigen Compilers. Dieses Compiler-Programm ist ein Muster-Beispiel für eine systematische Entwicklung eines gut strukturierten Programmes nicht-trivialer Komplexität. Hauptsächlich aber soll dieses Kapitel als Einführung in die Technik des Compilerbaus dienen. Kenntnisse und Einblick in dieses Thema fördern ganz allgemein die Fähigkeiten des Programmierens in höheren Programmiersprachen. Ferner sind sie Grundlage und Vorbedingung für die Möglichkeit, eigene Eingabesprachen und Systeme für spezifische Zwecke zu konstruieren. Da der Compilerbau ein komplexes Thema mit vielen Aspekten ist, beschränken wir uns in dieser Hinsicht auf eine Einführung. Vielleicht die wichtigste Grundidee ist dabei die Erkenntnis, dass die grammatikalische Struktur einer Sprache sich in der Struktur des Compilers widerspiegeln muss. Daraus folgt unmittelbar, dass die Komplexität - oder besser die Einfachheit - einer Sprache das Mass für die Komplexität ihres Compilers darstellt. Wir beginnen daher mit einer Behandlung von Sprachstrukturen und deren formeller Beschreibung im allgemeinen. Danach konzentrieren wir uns ausschliesslich auf einfache Strukturelemente, die sich durch einfache, effiziente und übersichtliche Compiler behandeln lassen. Die Erfahrung hat gezeigt, dass Elemente von struktureller Einfachheit durchaus genügen, um praktisch alle echten Bedürfnisse zu erfüllen, die an Programmiersprachen gestellt werden.

Die zwei letzten Kapitel sind praktischen Problemen des Compiler-Baus gewidmet. Anhand einer Fall-Studie wird gezeigt, wie eine bestehende Sprache und ihr Compiler erweitert werden. Dieser in der Praxis häufige Fall wird durch eine systematische Struktur des zu erweiternden Systems ganz wesentlich erleichtert. Letzlich werden verschiedene Methoden der Uebertragung eines Compilers auf fremde Computer dargelegt. Sie bauen alle auf der Technik des *bootstrapping* auf und bedingen, dass der Compiler in seiner eigenen Sprache beschrieben wird.

## 1. Definition und Struktur formaler Sprachen

Jeder Sprache liegt ein *Vokabular* zugrunde. Normalerweise bezeichnet man seine Elemente als Worte. Bei formalen Sprachen hingegen ist es üblich, sie *Symbole* oder *Grundsymbole* zu nennen. In jeder Sprache gibt es Folgen von diesen Symbolen, die als korrekt oder wohlgeformt, andere die als falsch oder missgebildet gelten. In erster Linie ist es die Grammatik oder *Syntax*, die bestimmt, zu welcher Kategorie eine Symbolfolge gehört. Wir gehen hier sogar soweit, dass wir die Menge von Symbolfolgen, die von der Syntax als wohlgeformt definiert sind, als die Sprache selbst bezeichnen. Missgebildete Folgen gehören überhaupt nicht zur Sprache, auch wenn sie ausschliesslich aus Symbolen des zugehörigen Vokabulars aufgebaut sind.

Die erste Funktion der Syntax ist also die Beschreibung der Menge der Symbolfolgen, genannt *Sätze*, die zur Sprache gehören. Die zweite, nicht minder wichtige Funktion ist die Definition einer Satzstruktur. Diese spielt eine eminente Rolle in der Erkennung der Bedeutung eines Satzes. Dies geht schon daraus hervor, dass Mehrdeutigkeiten meistens darauf beruhen, dass derselben Wortfolge mehrere korrekte Satzstrukturen zugeordnet werden können. Struktur (Syntax) und Bedeutung (Semanik) sind daher eng miteinander verbunden. Die Syntax dient letzlich immer einem höheren Ziel, nämlich einer Codierung einer bestimmten Bedeutung. Dies soll uns aber nicht daran hindern, zunächst ausschliesslich syntaktische Aspekte zu behandeln und die Probleme der Bedeutung und deren Interpretation zu ignorieren.

Zur Einführung wollen wir als Beispiel den Satz *Katzen schlafen* betrachten. Das Wort *Katzen* ist das Subjekt und schlafen das Praedikat des Satzes. Dieser Satz gehört zur Sprache, die zum Beispiel durch folgende Syntax definiert ist:

Satz = Subjekt Praedikat.
Subjekt = Katzen | Hunde.
Praedikat = essen | schlafen.

Die Bedeutung dieser drei Zeilen ist die folgende:

1. Ein Satz besteht aus einem Subjekt gefolgt von einem Praedikat.
2. Ein Subjekt besteht aus dem Wort *Katzen* oder dem Wort *Hunde*.
3. Ein Praedikat besteht aus dem Wort *essen* oder dem Wort *schlafen*.

Die Grundidee ist, dass jeder (wohlgeformte) Satz durch wiederholte Anwendung von Substitutionsregeln aus dem Startsymbol (*Satz*) hergeletet werden kann. Als Notation verwenden wir im folgenden eine Variante der sogenannten Backus-Naur Form (BNF), die zuerst zur Definition der Programmiersprache Algol 60 verwendet wurde. Die Satzgefüge *Satz*, *Subjekt*, und *Praedikat* nennt man *nicht-terminale* Symbole, die Worte, die in den eigentlichen Sätzen vorkommen, heissen *terminale* Symbole, und die Substitutionsregeln heissen *Produktionen*. Die Zeichen = | und der Punkt sind

sogenannte *Metasymbole* der BNF Notation. Zur Benennung von nicht-terminalen Symbolen verwenden wir sinnvoll gewählte Worte, die auf die Bedeutung der entsprechenden Strukur hinweisen sollen. Wenn anstatt der deskriptiven Worte im obigen Beispiel lediglich einzelne Buchstaben eingesetzt werden, wie dies in mathematischen Formeln üblich ist, so erhalten wir folgende, formal beschriebene Syntax. Zur Unterscheidung verwenden wir Grossbuchstaben für nicht-terminale Symbole, Kleinbuchstaben für Terminalsymbole.

*Beispiel 1*

$$\begin{aligned} S &= AB. \\ A &= x \mid y. \\ B &= z \mid w. \end{aligned} \qquad (1)$$

Die definierte Sprache besteht aus den vier Sätzen xz, yz, xw, yw.

Die bisher informell eingeführten Konzepte sollen nun durch folgende Definitionen genau festgelegt werden. Wir bezeichnen einzelne Symbole durch lateinische, Symbolfolgen durch griechische Buchstaben.

1. Eine Sprache wird charakterisiert durch ein Vier-Tupel L(T,N,P,S).
   Das Vokabular T von Terminalsymbolen.
   Die Menge N von nicht-terminalen Symbolen.
   Die Menge P von Substitutionsregeln, genannt Produktionen.
   Das Startsymbol S, ein Element von N.

2. Die *Sprache* L(T,N,P,S) ist die Menge der Folgen von Terminalsymbolen, die aus dem Startsymbol S nach den folgenden Regeln (siehe 3.) hergeleitet werden können.

$$L = \{\xi \mid S \overset{*}{\rightarrow} \xi \ \ \& \ \ \xi \in T^*\}$$

$T^*$ bezeichnet die Menge der Symbolfolgen, die aus Symbolen des Vokabulars T aufgebaut werden können.

3. Eine Folge $\sigma_n$ kann aus einer Folge $\sigma_0$ *hergeleitet* werden, wenn und nur wenn es Folgen $\sigma_1, \sigma_2, \ldots, \sigma_{n-1}$ gibt, so dass jedes $\sigma_i$ aus dem vorangehenden $\sigma_{i-1}$ direkt hergeleitet werden kann (siehe 4.). Wir schreiben:

$$\sigma_0 \overset{*}{\rightarrow} \sigma_n \quad \equiv \quad \sigma_{i-1} \rightarrow \sigma_i \qquad \text{für } i = 1 \ldots n$$

4. Eine Folge $\eta$ kann aus einer Folge $\xi$ *direkt hergeleitet* werden, wenn und nur wenn sich $\eta$ und $\xi$ als Folgen $\eta = \alpha\eta'\beta$ und $\xi = \alpha\xi'\beta$ darstellen lassen und $\eta' = \xi'$ eine Produktion in der Menge P ist.
Man beachte, dass die Notation $\alpha = \beta_1|\beta_2| \ldots |\beta_n$ lediglich eine Abkürzung für die Menge der Produktionen $\alpha = \beta_1, \alpha = \beta_2, \ldots, \alpha = \beta_n$ darstellt.

Nach diesen Ausführungen kann zum Beispiel der Satz xz aus Beispiel 1 durch folgende Reihe von Schritten hergeletet werden:

$$S \rightarrow AB \rightarrow xB \rightarrow xz$$

Daraus folgt, dass $S \overset{*}{\rightarrow} xz$, und weil $xz \in T^*$, so ist xz ein Element der Sprache, d.h. $xz \in L$.

Man beachte, dass die nicht-terminalen Symbole A und B nur in Zwischenschritten vorkommen, und dass der letzte Schritt zu einer Folge führt, die nur terminale Symbole enthält. Die syntaktischen Regeln werden Produktionen genannt, weil sie bestimmen, wie ein Satz produziert weden kann.

Eine Sprache heisst (nach N. Chomsky) *kontext-frei*, wenn (und nur wenn) sie durch eine Menge von kontext-freien Produktionen definiert werden kann. Eine Produktion ist kontext-frei, wenn (und nur wenn) sie die Form

$$A = \sigma. \qquad (A \in N, \ \sigma \in (N \cup T)^*)$$

hat, das heisst, wenn ihre linke Seite aus einem einzigen Symbol besteht. Dies bedeutet nichts weniger, als dass A durch $\sigma$ ersetzt werden kann, ungeachtet des Kontextes, in dem A vorliegt. Falls eine Produktion die Form

$$\alpha A \beta = \alpha \sigma \beta.$$

hat, so heisst sie *kontext-abhängig*. Sie besagt nämlich, dass A nur durch $\sigma$ ersetzt werden kann, sofern A in der Umgebung von $\alpha$ und $\beta$ vorliegt. Wir werden uns im folgenden ausschliesslich mit kontext-freien Sprachen beschäftigen.

Das zweite Beispiel einer Syntax zeigt, wie durch die Verwendung von *Rekursion* eine unendliche Menge von Sätzen durch eine endliche Menge von Produktionen beschrieben werden kann.

*Beispiel 2*

$$S = xA.$$
$$A = z \mid yA.$$

Aus dem Startsymbol lassen sich die folgenden Sätze herleiten:

xz, xyz, xyyz, xyyyz, xyyyyz, xyyyyyz, ...

Das Beispiel zeigt also, wie durch Anwendung einer rekursiven Definition beliebige *Repetitionen* erzeugt werden können. Allerdings ist die Rekursivität nicht unmittelbar ersichtlich. Wir führen daher zur leichteren Sichtbarmachung einer Repetition eine zusätzliche Notation ein:

$$\{ \sigma \}$$

bedeute, dass die Folge $\sigma$ beliebig viele Male (inklusive 0 mal) repetiert werden kann. $\{\sigma\}$ ist also gleichbedeutend mit $\varepsilon \mid \sigma \mid \sigma\sigma \mid \sigma\sigma\sigma \mid \ldots$, und die zwei Produktionen aus Beispiel 2 lassen sich zusammenfassen in die eine Produktion

$$S = x\{y\}z.$$

Ebenfalls im Sinne einer Verbesserung der Lesbarkeit führen wir die Notation

$$[ \sigma ]$$

ein. $[\sigma]$ ist gleichbedeutend mit $\sigma \mid \varepsilon$, wobei $\varepsilon$ die leere Folge bezeichnet. So

besagt zum Beispiel die Produktion

S = x[y]z.

dass sich aus S die Folgen xyz und xz herleiten lassen. Es erscheint nun als angezeigt, auch gewöhnliche Klammerausdrücke zuzulassen. Anstelle von xyz|xwz, kann dann zum Beispiel x(y|w)z geschrieben werden. Es sei noch darauf hingewiesen, dass die Klammern {} [] () Metasymbole wie die Zeichen = | und . sind. Sie gehöhren also zur erweiterten BNF Metasprache, und nicht zur Objektsprache, deren Syntax definiert wird.

## 2. Satzanalyse

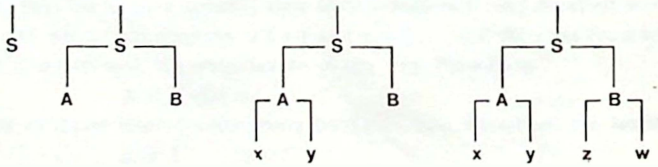
Die Aufgabe eines Uebersetzers oder Compilers ist normalerweise nicht die Erzeugung oder Herleitung, sondern die *Erkennung* eines Satzes und seiner Struktur. Dies bedeutet, dass die Schritte der Erzeugung beim Lesen eines Satzes als Schritte der Erkennung nachvollzogen werden müssen. Dies ist im allgemeinen eine komplizierte, und in vielen Fällen sogar eine unmögliche Aufgabe. Ihre Komplexität hängt unmittelbar von der Art der Produktionen ab, die die Syntax bestimmen. Zahlreiche Erkennungs-Algorithmen für verschiedene Klassen von Sprachen unterschiedlicher struktureller Komplexität sind bekannt. Ihre Effektivität hängt direkt von ihrer Mächtigkeit ab: je allgemeiner sie verwendbar sind, umso weniger Effizienz kann erwartet werden. Unser Ziel hier ist jedoch lediglich die Entwicklung einer Methodik, relativ einfach Sprachen zu erkennen, dies jedoch mit einer Effizienz, wie sie von der Praxis gefordert wird. Die Zeit, die gebraucht wird, um einen Satz zu erkennen, darf demnach höchstens proportional zu seiner Länge anwachsen. Anstatt einen möglichst allgemeinen Erkennungs-Algorithmus, genannt *Parser*, zu finden, gehen wir in umgekehrter Richtung vor: Wir postulieren einen einfachen Algorithmus und bestimmen danach die Klasse der Sprachen, die er verarbeiten kann.

Eine erste Folge dieser grundlegenden Forderung ist, dass jeder Schritt in der Zerlegung eines Satzes lediglich durch den gegenwärtigen Zustand der Analyse und ein einziges, nächstes zu lesendes Symbol bestimmt werden darf. Ferner wird gefordert, dass kein Schritt später rückgängig gemacht werden darf. (*one symbol lookahead, no backtracking*).

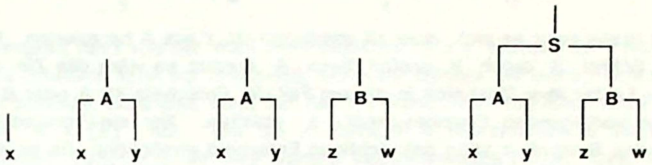
Die grundlegende Methodik, die nachfolgend beschrieben wird, beruht auf dem sogenannten *top-down* Prinzip. Es ist zweckmässig, die Struktur eines Satzes als Baum darzustellen. Zuoberst steht das Startsymbol, an seiner Basis der zu erkennende Satz. Zur Illustration möge folgendes Beispiel dienen:

S = AB.
A = xy.
B = zw.

Beim Lesen (von links nach rechts) durchläuft der Parser die folgenden Schritte, wobei von S ausgehend eine Struktur aufgebaut wird, die an der Basis dem zu lesenden Satz entspricht. Der Baum wächst also von oben nach unten; daher der Ausdruck *top-down*.



Umgekehrt wird beim sogenannten *bottom-up* Prizip vorgegangen. Anstatt "spekulativ" vom Startsymbol auszugehen und eine Folge zu *produzieren*, die dem eingelesenen Text entspricht, wird der eingelesene Text *reduziert*, bis zuletzt der ganze Text in das Startsymbol zurückgeführt ist. In der graphischen Darstellung wächst der Strukturbaum von unten nach oben; daher die Bezeichnung *bottom-up.*




Anhand des Beispiels 1 soll nun gezeigt werden, wie ein einfacher top-down Parser vorgeht. Wir nehmen an, dass er den Satz *xw* zu erkennen habe. *xw* gehört nur dann zur Sprache, wenn es aus dem Startsymbol S hergeleitet werden kann. Aus S ist jedoch nur die Folge AB direkt herleitbar. Wir ersetzen daher S auf jeden Fall durch AB. Nun muss untersucht werden, ob sich der Anfang des Satzes xw aus AB herleiten lässt. In der Tat bestätigt die Produktion A = x, dass dies möglich ist; x lässt sich also, zusammen mit A, als erledigt abstreichen. Es bleibt noch zu zeigen, dass w sich aus B herleiten lässt. Beim Durchsuchen der Syntax erkennen wir die Produktion B = z als nicht anwendbar; hingegen bringt uns die Produktion B = w ans Ziel. Wir können uns die einzelnen Schritte wie folgt aufzeichnen, wobei wir links die Symbolfolge auftragen, aus welcher der noch verbleibende Rest des zu erkennenden Satzes herleitbar sein muss. Rechts steht der Rest des Satzes selbst.

| | |
|---|---|
| S | xw |
| AB | xw |
| <u>x</u>B | <u>x</u>w |
| B | w |
| <u>w</u> | <u>w</u> |

Man beachte, dass in diesem Beispiel jeder Schritt sich eindeutig aus dem zunächst zu verfolgenden Ziel (z.B. A) und dem nächsten vorliegenden Symbol (z.B. x) bestimmen lässt. Dies ist leider nicht in allen Fällen zutreffend, wie das folgende Beispiel einer Syntax zeigt.

*Beispiel 3*

S = A | B.
A = xA | y.
B = xB | z.

Wir versuchen, den Satz xxz (der tatsächlich zur definierten Sprache gehört) zu erkennen.

| | |
|---|---|
| S | xxz |
| A | xxz |
| xA | xxz |
| A | xz |
| xA | xz |
| A | z |

An dieser Stelle zeigt es sich, dass es unmöglich ist, z aus A herzuleiten. Hätten wir im ersten Schritt S durch B anstatt durch A ersetzt, so wäre das Ziel erreichbar gewesen. Leider aber lässt sich in diesem Fall der Entscheid für A oder B nicht aus dem einen vorliegenden Eingabesymbol x ableiten. Nur ein Vorausblicken und Erkennen des Symbols z hätte den richtigen Entscheid ermöglicht. Es ist aber leicht ersichtlich, dass in diesem Beispiel einer Syntax keine feste Grenze für die Anzahl der zu betrachtenden Symbole gegeben werden kann. Denn jeder Algorithmus, der fähig wäre, über n Symbole vorauszublicken, könnte mit einer Folge von n Symbolen x gefolgt von einem z (oder y) zu Fall gebracht werden. Anstatt nun aber einen komplizierteren Erkennungs-Algorithmus vorzuschlagen, wollen wir eine Regel postulieren, die die Syntax einschränkt, indem sie den obigen Fall ausschliesst. Wir nehmen an, dass stets alle Produktionen mit demselben Linkssymbol in eine einzige Regel zusammengefasst vorliegen. Zudem benötigen wir

**Definition 1**

Wir bezeichnen die Menge aller Symbole, die am Anfang einer Folge stehen können, die aus einer Folge $\sigma$ herleitbar ist, mit $first(\sigma)$.

$$first(\sigma) = \{s \mid \sigma \overset{*}{\rightarrow} s\sigma'\}$$

**Regel 1**

Für jede Produktion

$$A = \sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_n$$

wird verlangt, dass die Initialsymbolmengen aller Terme $\sigma_i$ disjunkt sind.

$$first(\sigma_i) \cap first(\sigma_j) = \emptyset \qquad \text{für alle } i \neq j$$

In der Syntax von Beispiel 3 gilt $x \in first(A)$ und $x \in first(B)$. Regel 1 ist also durch die Produktion $S = A \mid B$ verletzt. Nun ist es in diesem Beispiel allerdings leicht, diese Schwierigkeit zu umgehen, indem eine aequivalente Syntax definiert wird, welche Regel 1 beachtet:

$$S = C \mid xS.$$
$$C = y \mid z.$$

Leider genügt Regel 1 noch nicht, um alle Schwierigkeiten auszuschliessen, wie das folgenden Beispiel zeigt.

*Beispiel 4*

$$S = Ax.$$
$$A = x \mid \varepsilon .$$

$\varepsilon$ bezeichne wiederum die leere Folge. Falls wir nun versuchen, den Satz x zu erkennen, geraten wir erneut in eine Sackgasse:

| | |
|---|---|
| S | x |
| Ax | x |
| $\underline{x}$x | $\underline{x}$ |
| x | $\varepsilon$ |

Die Schwierigkeit rührt von der Wahl der Produktion A = x anstelle der Produktion A = $\varepsilon$ Diese Situation kann nur entstehen, wenn ein Symbol die leere Folge erzeugen kann. Man beachte, dass dies stets der Fall ist, wenn eine Produktion die Form A = {s} oder A = [s] hat. Um die in Beispiel 4 charakterisierte Schwierigkeit auszuschliessen, adoptieren wir eine weitere einschränkende Regel. Dazu benötigen wir

**Definition 2**

Die Menge aller Symbole, die einer aus $\sigma$ hergeleiteten Folge unmittelbar nachfolgen können, wird mit *follow*($\sigma$) bezeichnet.

$$follow(\sigma) = \{s \mid S \overset{*}{\rightarrow} \alpha\sigma' s\beta \ \& \ \sigma \overset{*}{\rightarrow} \sigma'\}$$

**Regel 2**

Für jedes nicht-terminale Symbol A, aus welchen die leere Folge hergeleitet werden kann, muss die Menge seiner Initialsymbole disjunkt von der Menge der Folgesymbole sein.

$$A \overset{*}{\rightarrow} \varepsilon \quad \supset \quad first(A) \cap follow(A) = \emptyset$$

In Beispiel 4 wird die Regel 2 durch das Symbol A verletzt, indem

$$first(A) = follow(A) = \{x\} \neq \emptyset$$

Man beachte auch, dass Regel 4 durch jede linksrekursive Produktion der Form

$$A = A\xi \mid \varepsilon .$$

verletzt wird, da *first*(A) = *follow*(A) = *first*($\xi$) $\neq \emptyset$ . Linksrekursive Produktionen sind daher durch die obigen Restriktionsregeln ausgeschlossen.

Aus den vorangehenden Ausführungen ist man vielleicht versucht, den Schluss zu ziehen, dass Schwierigkeiten mit syntaktischen Regeln stets dadurch zu lösen sind, dass eine aequivalente, den Regeln genügende Syntax konstruiert wird. Es ist daher angezeigt, an dieser Stelle daran zu erinnern, dass die Syntax nur Mittel zu einem höheren Zweck ist, nämlich zur Sichtbarmachung der Bedeutung eines Satzes. Wird nun eine Syntax umgeformt, so muss stets beachtet werden, dass damit die Bedeutungsstruktur der Sprache nicht in Mitleidenschaft gezogen wird. Die folgenden Produktionen definieren z.B. eine Form von Ausdrücken mit Operanden a, b, c, und dem Operator "-", den wir als Subtraktion auffassen wollen.

$$S = A \mid S\text{-}A.$$

A = a | b | c.

Da die linksrekursive Produktion für S Regel 1 verletzt, versuchen wir, eine aequivalente Syntax zu erstellen, und finden die Lösung als

S = AB.
B = [-S].
A = a | b | c.

Während nun aber die erste Version dem Satz a-b-c die Struktur zuweist, die durch Klammerung ((a-b)-c) hervorgehoben werden kann so erhält der gleiche Ausdruck von der zweiten Version implizit die Struktur (a-(b-c)) zugewiesen. Unter Annahme der üblichen Regeln der Subtraktion ist es leicht ersichtlich, dass die beiden Versionen zwar syntaktisch, nicht aber semantisch aequivalent sind. Die Lehre, die wir aus dem Beispiel ziehen, ist, dass bei der Umformung einer Syntax stets auch die *Bedeutungsstruktur* der Sprache mitberücksichtigt werden muss.

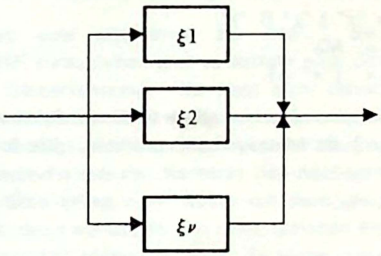
## 3. Syntax Graphen

Die Darstellung einer Syntax in BNF ist nur eine von verschiedenen Möglichkeiten. Eine andere, in vieler Hinsicht vorteilhafte Art der Darstellung beruht auf der Verwendung von Diagrammen oder *Graphen*. Der Hauptvorteil beruht dabei auf der besseren Ueberschaubarkeit. Wir schlagen nachfolgend eine Art der Graphen vor, die den Ablauf einer Satzerkennung im *top-down* Verfahren unmittelbar veranschaulicht. Wir geben ein einfaches Rezept, wie eine durch BNF definierte Syntax konsequent in entsprechende Graphen übersetzt werden kann. Der umgekehrte Vorgang ist natürlich ebenso leicht möglich. Wir nehmen an, dass die BNF bereits in dem Sinne normalisiert ist, dass jedes nicht-terminale Symbol durch eine einzige Produktionsfolge definiert ist. Die Uebersetzung ist dann durch nachfolgende Regeln bestimmt.

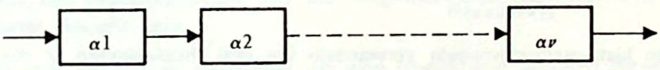
A1. Jedes nicht-terminale Symbol, definiert durch eine Produktionenfolge

$$A = \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n.$$

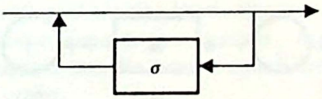
wird als Graph mit der folgenden Struktur abgebildet:



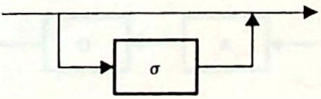

A2. Jeder Term $\alpha_1 \alpha_2 \ldots \alpha_n$ wird in einen Graphen der folgenden Struktur abgebildet:



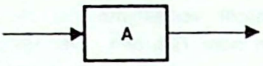

A3. Hat ein Element die Form $\{\sigma\}$ , so wird es in die folgende Graphenstruktur abgebildet:



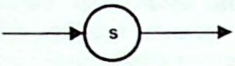

A4. Hat ein Element die Form $[\sigma]$ , so wird es in die folgende Struktur abgebildet:



**A5.** Ist ein Element ein nicht-terminales Symbol A, so wird es durch ein Rechteck eingefasst.




**A6.** Ist ein Element ein Terminalsymbol s, so wird es durch einen Kreis gekennzeichnet.




*Beispiel 5*

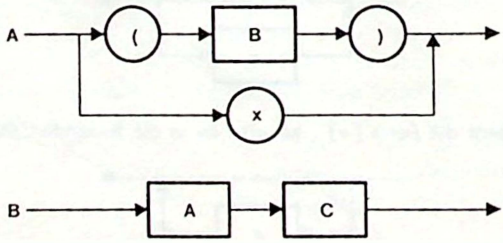
Wir adoptieren im nachfolgenden Beispiel die Konvention, dass alle ~~nicht~~ terminalen Symbole durch Einklammerung mittels Gänsefüsschen eindeutig erkennbar gemacht werden. Diese Notation verwenden wird für alle späteren Beispiele.

```
A = "x" | "(" B ")".
B  =  AC.
C  =  {"+" A}.
```

Die Zeichen +, (, ), und x sind also hier ~~nicht~~ terminale Symbole der zu definierenden Sprache, während { und } als Metasymbole auftreten. Die folgenden Sätze gehören z.B. zu dieser Sprache:

```
x
(x)
(x+x)
((x))
((x+(x+x)))
```

Die obigen Uebersetzungsregeln verwandeln die drei Produktionen in die in Fig.1 abgebildeten Graphen. Indem C in B und B in A direkt substituiert werden, ergibt sich als Zusammenfassung der einzige, in Fig.2 dargestellte Graph.



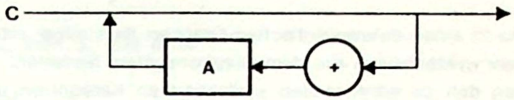



Fig. 1. Syntax nach Beispiel 5

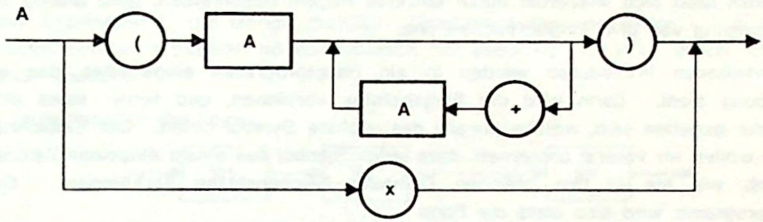



Fig. 2. Reduzierter Graph nach Beispiel 5

Die Graphen sind also eine alternative zur BNF. Sie sind in Bezug auf Uebersichtlichkeit der BNF vorzuziehen und vermitteln eine unmittelbarere Vorstellung über den Prozess der Satzerkennung. Es fragt sich daher, wie sich die beiden postulierten Regeln im graphischen Bild äussern. In der Tat wird ihr Sinn dabei erst recht deutlich. *Regel 1* besagt nämlich, dass bei jeder Verzweigung im Graphen der zu beschreitende Ast eindeutig aus der Kenntnis des nächsten Symbols hervorgehen muss. Dies impliziert, dass keine zwei Aeste mit dem gleichen Symbol beginnen dürfen. *Regel 2* verlangt, dass ein Graph, der ohne Einlesen eines Symbols traversiert werden kann, als "durchsichtig" betrachtet wird. Es muss eindeutig entscheidbar sein, ob das nächste Symbol zu dem Graphen oder bereits zu seinem Nachfolger gehört. Die Mengen der Anfangssymbole und der möglicherweise nachfolgenden Symbole müssen daher disjunkt sein.

Es ist nun sehr einfach zu verfizieren, ob eine Menge von Graphen diesen beiden Regeln genügt. Es ist dabei durchaus nicht notwendig, auf die Syntax in BNF zurückzugreifen. Im Gegenteil erweist sich die übersichtliche graphische Darstellung als besonders geeignet. Vorsorglicherweise bestimmen wir zuerst wiederum für jedes nicht-terminale Symbol A seine zugehörigen Mengen *first*(A) und *follow*(A). Wir nennen ein System von Graphen, das die beiden einschränkenden Regeln erfüllt, ein *deterministisches* Graphensystem.

## 4. Aufbau eines Parsers für eine gegebene Syntax

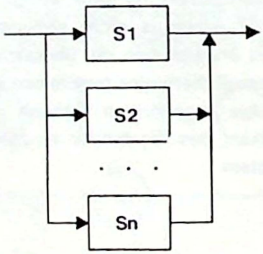
Falls eine Syntax durch einen deterministischen Graphen darstellbar ist, so lässt sich dieses Programm sehr systematisch aus dem Graphensystem herleiten. Die einzelnen Graphen entsprechen den zu erkennenden syntaktischen Kategorien und werden in einzelne Prozeduren abgebildet. Jeder Graph stellt sozusagen das Flussdiagramm der entsprechenden Prozedur dar. Die Uebersetzung des Graphensystems in ein Programm lässt sich wiederum durch einzelne Regeln beschreiben, ganz analog zur Uebertragung von BNF in graphische Form.

Die erhaltenen Prozeduren werden in ein Hauptprogramm eingebettet, das als Umgebung dient. Darin wird die Eingabefolge vereinbart, und ferner muss eine Prozedur gegeben sein, welche jeweils das nächste Symbol liefert. Der Einfachheit halber wollen wir vorerst annehmen, dass jedes Symbol aus einem einzelnen Zeichen besteht, wie sie in den üblichen Computer-Zeichensätzen vorkommen. Das Hauptprogramm wird also stets die Form

```
program Parser(input, output);
   var ch: char;
{Vereinbarungen der einzelnen Prozeduren}
begin read(ch); S
end .
```

haben, wobei S die dem Startsymbol S entsprechende Prozedur darstellt. *(read(ch)* dient als Kurzform für *read(input,ch))*. Die einzelnen Uebersetzungsschritte sind durch die folgenden Regeln B1 bis B6 bestimmt. Es ist dabei vorteilhaft, dass zuerst die Anzahl der nicht-terminalen Symbole reduziert wird, indem wenn möglich einzelne Graphen direkt in anderen Graphen eingesetzt werden, wie dies im obigen Beispiel 4 gezeigt wurde. Wir bezeichnen das Programm, das aus der Uebersetzung eines Graphen S hervorgeht, mit P(S).

B1. Jede Struktur mit der Form

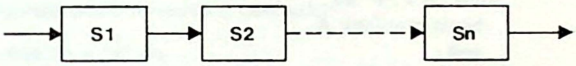



wird in eine bedingte oder selektive Anweisung übersetzt:

```
if ch in L1 then P(S1) else
if ch in L2 then P(S2) else
...............
if ch in Ln then Sn else error

case ch of
  L1: P(S1);  L2: P(S2);  ...   ; Ln: P(Sn)
end
```

Dabei bezeichnet $L_i$ die Menge *first*($S_i$). (Programmierhinweis: Wenn L aus einem einzigen Symbol s besteht, so soll natürlich der Ausdruck ch in L durch ch = s ersetzt werden).

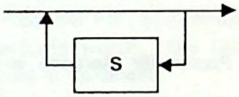
B2. Jede Struktur der Form




wird in eine Anweisungsfolge übersetzt:

```
begin P(S1); P(S2);  ...   ; P(Sn) end
```

B3. Jede Struktur der Form

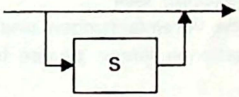



wird in eine repetitive Anweisung übersetzt:

```
while ch in L do  P(S)
```

L stellt die Menge *first*(S) dar (siehe Hinweis unter B1).

B4. Jede Struktur der Form




wird in eine bedingte Anweisung übersetzt:

```
if ch in L then  P(S)
```

B5. Jede Referenz zu einem Graphen (rechteckige Figur) wird in einen Aufruf der dem Graphen entsprechenden Prozedur übersetzt.

B6. Jede Referenz zu einem Terminalsymbol x (runde Figur) wird wie folgt in eine Leseanweisung übersetzt:

```
if ch = "x" then read(ch) else error
```

wobei *error* eine nicht näher spezifizierte Prozedur darstellt, die aufgerufen wird,

sobald eine inkorrekte Symbolfolge entdeckt wird.

Die Anwendung dieser Regeln soll nun am Beispiel 5 gezeigt werden. Sie resultiert im folgenden *Programm 1.* Dabei wurden allerdings bereits einige offensichtliche Vereinfachungen vorgenommen.

```
program Parser(input, output);
  var ch: char;

  procedure A;
  begin
      if ch = "x" then read(ch) else
      if ch = "(" then
      begin read(ch); A;
            while ch = "+" do
              begin read(ch); A
              end ;
            if ch = ")" then read(ch) else error
      end else error
  end ;

begin read(ch); A
end .
```

Programm 1: Parser für Sprache aus Beispiel 5

Eine buchstabengetreue Uebersetzung hätte z.B. zu folgender Anweisung geführt.

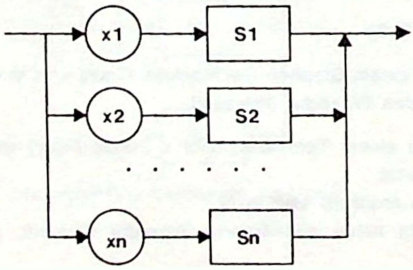
```
if ch = "x" then
      if ch = "x" then read(ch) else error
else ...
```

welche offensichtlich in die Anweisung

```
if ch = "x" then read(ch) else ...
```

vereinfacht werden kann. Solche Vereinfachungen sind tatsächlich relativ oft möglich. Zwei häufig vorkommende Situationen führen zu den folgenden Zusatzregeln:

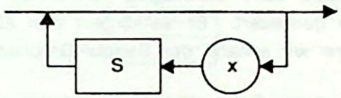
B1a. Eine Struktur der Form



wird in folgende Anweisung übersetzt:

```
if ch = "x1" then begin read(ch); P(S1) end else
if ch = "x2" then begin read(ch); P(S2) end else
        . . . . . . .
if ch = "xn" then begin read(ch); P(Sn) end else error
```

B3a. Eine Struktur der Form




wird in die folgende while-Anweisung übersetzt:

```
while ch = "x" do
        begin read(ch); P(S) end
```

Die Prozedur *error* bleibt vorläufig unbestimmt. Solange wir lediglich daran interessiert sind, herauszufinden, ob eine gegebene Zeichenfolge syntaktisch legitim sei oder nicht, genügt es, wenn beim Aufruf von *error* der Prozess abgebrochen wird. In der Praxis ist dies natürlich nicht der Fall, da normalerweise erwartet wird, dass der nachfolgende Text weiter untersucht werde. Die daraus hervorgehenden Probleme und Techniken werden im Kapitel 9 behandelt.

## 5. Tabellen-gesteuerte Syntax Analyse

Im vorangehenden Kapitel wurde eine Methodik beschrieben, die es erlaubt, für irgend eine Syntax den zugehörigen Parser zu programmieren, sofern die Syntax den zwei einschränkenden Regeln genügt. Da jedoch immer dasselbe Prinzip der Analyse verwendet wird, ist es auch möglich, ein einziges, allgemeines Programm zu erstellen, das für alle entsprechenden Sprachen geeignet ist. Dieses allgemeine Programm wird je nach Sprache mit Tabellen versehen, welche deren Syntax in codierter Form darstellen. Der Parser wird dann sozusagen von diesen - während der Analyse unveränderten - Tabellen *gesteuert*. Er verkörpert den Algorithmus, nach dem wir einen Satz erkennen, wenn wir anhand der Syntax-Graphen vorgehen.

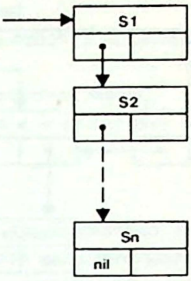
Die Syntax muss also in einer Form dargestellt werden, wie sie einem Computer Programm zugänglich ist. Weder BNF noch Graphen sind dafür sehr geeignet. Es ist naheliegend, von den Graphen als Fluss-Diagramme auszugehen, und diese in eine geeignete Daten-Struktur zu verwandeln. Dabei wird kaum eine einfache Tabellenform (Array) in Frage kommen, sondern viel eher eine Listenstruktur ohne festes, vorgegebenes Format. Den einzelnen Symbolen entsprechen Daten-Elemente (Knoten), während die Pfade (Kanten) durch Zeiger (pointers) dargestellt werden.

In der Programmiersprache *Pascal* ist der Record die geeignete Struktur, um Knoten in Listen darzustellen. Im vorliegenden Fall werden in jedem Datenelement vier Felder benötigt. Das erste Feld stellt das Symbol dar, welches dem Knoten entspricht. Zwei weitere Felder enthalten Zeiger, die entweder auf den nachfolgenden Knoten oder die Alternative verweisen, falls diese existieren. Ein weiteres Feld dient zur Angabe, ob es sich um ein terminales oder ein nicht-terminales Symbol handelt. Diese Record-Struktur wird vorteilhafterweise als Record-Varianten-Typ definiert, wobei das Artfeld als *tagfield* auftritt.

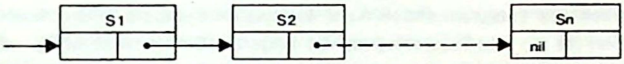
```
type pointer = ↑node;
  node = record suc, alt: pointer;
              case terminal: Boolean of
                true: (tsym: char);
                false: (nsym: hpointer)
           end
```

Es liegt nun auf der Hand, die Uebersetzung eines Systems von Graphen in eine entsprechende Datenstruktur wiederum durch eine Liste von festen Regeln zu definieren. Danach ist es möglich, sogar ein Programm zu konstruieren, welches diese Umformung automatisiert. Die Regeln sind in der Tat den bisher angegebenen Regeln sehr ähnlich.

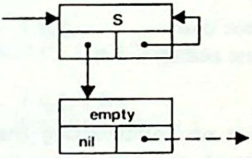
C1. Eine Graphenstruktur, wie sie unter B1 angegeben ist, wird in folgende Listenstruktur abgebildet:



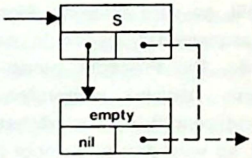

C2. Eine Graphenstruktur, wie sie unter B2 angegeben ist, wird in folgende Listenstruktur abgebildet:




C3. Eine Schleifenstruktur (siehe B3) wird wie folgt abgebildet:




C4. Die unter B4 angegebene Struktur wird wie folgt dargestellt:




Zur Veranschaulichung verwenden wir nochmals Beispiel 5 (siehe Fig.2). Nach obigen Regeln wird dieser Graph in folgende Datenstruktur umgewandelt.

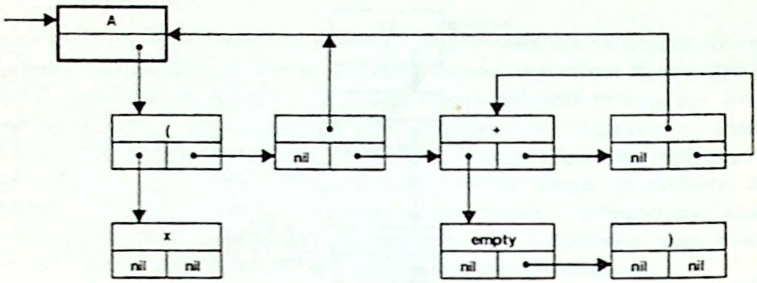



Fig. 3. Datenstruktur entsprechend dem Syntaxgraphen der Fig.2

Die obige Datenstruktur wird durch einen sogenannten *Leitknoten* identifiziert. Er enthält den Namen des entsprechenden nicht-terminalen Symbols in druckbarer Form. Diese Information ist für die Satzzerlegung im engeren Sinne nicht nötig. Sie wird aber im später zu beschreibenden Algorithmus verwendet, um den Ablauf der Zerlegung sichtbar zu machen. Der Datentyp *hpointer* wird dabei wie folgt vereinbart:

```
type hpointer = ↑header;
    header = record symbol: char;
                    entry: pointer
             end
```

Der zu entwerfende Algorithmus zur Satzzerlegung (Parser) besteht nun aus einer repetierten Anweisung, die besagt, wie von einem Knoten der Datenstruktur zum nächsten übergegangen wird. Das Programm beschreibt also, wie ein Graph, dargestellt als Listenstruktur, *interpretiert* wird. Falls ein Knoten erreicht wird, der ein nicht-terminales Symbol darstellt, so wird derselbe Algorithmus auf die dem Symbol entsprechende Datenstruktur angewendet, bevor in der gegenwärtig vorliegenden Struktur weitergeschritten wird. Die Prozedur *parse*, die das Traversieren einer Struktur beschreibt, wird also *rekursiv* aufgerufen. Falls ein Knoten einem Terminalsymbol darstellt, so wird geprüft, ob das nächste Eingabesymbol (sym) damit identisch ist. Ist dies der Fall, so wird dieses Symbol (Zeichen) eingelesen und zum Nachfolger übergegangen (suc). Ansonst wird die Alternative (alt) inspiziert, falls eine existiert.

```
procedure parse(goal: hpointer;  var match: Boolean);
   var s: pointer;
begin s := goal↑.entry;
   repeat
      if s↑ terminal then
         begin if s↑.tsym = sym then
```

```
                begin match := true; getsym
                end
            else match := (s↑.sym = empty)
        end
    else parse(s↑.nsym, match);
    if match then s := s↑.suc else s := s↑.alt
  until s = nil
end
```

Das obige Programm besitzt die Eigenschaft, ein neues Ziel zu verfolgen, sobald ein solches durch ein vorliegendes nicht-terminales Symbol A angezeigt wird. Und zwar geschieht dies, ohne dass zuerst geprüft wird, ob das nächste Eingabesymbol zur Menge der Anfangssymbole *first*(A) gehöre. Dies bedingt aber, dass für jede Verzweigung in einem Syntax-Graphen gelten muss, dass höchstens ein einziger Ast mit einem nicht-terminalen Symbol beginnen darf. Falls ein Ast der leeren Folge entspricht, so darf sogar kein paralleler Ast mit einem nicht-terminalen Symbol beginnen. Wir formulieren diese Bedingungen nachfolgend als Einschränkungen der Form von Produktionen, und nehmen an, dass jedes nicht-terminale Symbol A durch eine einzige zusammengesetzte Produktion definiert ist. Diese muss eine der folgenden Formen haben:

1. $A = s_1\xi_1 \mid s_2\xi_2 \mid \ldots \mid s_n\xi_n$
   $A \in N,\ s_i \in T,\ \xi_i \in (NUT)^*,\ i \neq j \supset s_i \neq s_j$

2. $A = s_1\xi_1 \mid s_2\xi_2 \mid \ldots \mid s_n\xi_n \mid B\eta$
   $B \in N,\ \eta \in (NUT)^*,\ s_i \notin first(B)$ für $i = 1 \ldots n$

3. $A = s_1\xi_1 \mid s_2\xi_2 \mid \ldots \mid s_n\xi_n \mid \varepsilon$
   $s_i \notin follow(A)$ für $i = 1 \ldots n$

Weniger restriktive Regeln bedingen raffiniertere Algorithmen, die sich mit unterschiedlichem Aufwand realisieren lassen. Sogar von der "no backtracking" Regel kann abgewichen werden, indem das obige Programm entsprechend verallgemeinert wird. Dies bringt jedoch einen ins Gewicht fallenden Verlust an Effizienz mit sich, der für praktische Anwendungen möglichst vermieden werden sollte.

Die Darstellung einer Syntax mit Graphen hat neben Vorteilen einen entschiedenen Nachteil: sie kann nicht direkt von einem Computer eingelesen werden. In dieser Hinsicht erscheint die BNF als ideal, weil sie eine Syntax in konventioneller Form als Formeltext darstellt, der aus einzelnen Buchstaben und Zeichen besteht. Im folgenden Kapitel wird daher die Frage behandelt, wie Parser-Tabellen direkt aus einer BNF-Spezifikation erhalten werden.

## 6. Die Uebersetzung von BNF-Produktionen in Parser-Tabellen

Im Unterschied zu den vorangegangenen Beschreibungen von Darstellungs-Transformationen streben wir in diesem Kapitel nicht eine informelle Definition, sondern ein formales Programm an. Dieses Ziel erscheint nicht nur deshalb interessant, weil sich damit die Umformung durch Computer automatisch und fehlerfrei durchführen lässt, sondern ganz besonders weil dieses Transformations- Programm als unmittelbare Anwendung der in Kapitel 4 beschriebenen Technik erscheint.

Dieser Umstand erklärt sich aus der Tatsache, dass die BNF - oder genauer die hier verwendete erweiterte BNF - dazu verwendet werden kann, sich selbst zu definieren. Wir bezeichnen diese nachfolgend genauer zu definierende Metasprache als *Erweiterte Backus-Naur Form* (EBNF). Das Ziel ist also, einen Parser für diese Metasprache zu konstruieren. Wir gehen dabei in folgenden vier Schritten vor:

*Schritt 1*: Definition der Syntax der EBNF, ausgedrückt in EBNF selbst.

*Schritt 2*: Konstruktion eines Parsers für diese Syntax nach den Prinzipien aus Abschnitt 4.

*Schritt 3*: Ausbau dieses Parsers in einen Uebersetzer, der Parser-Tabellen erzeugt.

*Schritt 4*: Integration dieses Uebersetzers mit dem Tabellen-gesteuerten Parser aus Abschnitt 5.

**Schritt 1.**

In Uebereinstimmung mit bisher befolgten Notationsregeln soll der Eingabetext mit den in EBNF formulierten Produktionen folgende Bedingungen erfüllen.

1. Nicht-terminale Symbole bestehen aus einem einzigen Grossbuchstaben.
2. Terminalsymbole sind einzelne Zeichen (nicht notwendigerweise Buchstaben), gekennzeichnet durch Anführungszeichen.
3. Jede Produktion nimmt eine Zeile ein.

Die Syntax, die eine EBNF-Produktion beschreibt, ergibt sich sodann aus folgenden Produktionen. Die nicht-terminalen Symbole P, A, T, F können als Abkürzungen für *Produktion, Ausdruck, Term,* und *Faktor* aufgefasst werden. $\beta$ bezeichne einen beliebigen Grossbuchstaben, $\sigma$ ein beliebiges Zeichen.

P = $\beta$ "=" A "." .
A = T {"|" T}.
T = F {F}.
F = $\beta$ | """ $\sigma$ """ | "(" A ")" | "[" A "]" | "{" A "}" .

EBNF-Syntax dargestellt in EBNF

**Schritt 2**

Obwohl die obigen Regeln der lexicographischen Textdarstellung den Aufbau des Parsers im wesentlichen einfach gestalten, lässt sich doch das Einlesen des nächsten Symbols nicht direkt durch die Anweisung *read(ch)* darstellen. Zumindest sollten doch Leerzeichen in beliebiger Anzahl zwischen den Symbolen eingestreut werden dürfen. Für das Einlesen postulieren wir daher eine Prozedur *getsym*, welche der Variablen *sym* das nächste Symbol zuweist, ungeachtet möglicherweise eingestreuter Leerzeichen. Diese Prozedur stellt ein triviales Beispiel eines Mechanismus dar, der im allgemeinen ein *Scanner* genannt wird. Sein Zweck ist das Herausgreifen des nächsten Symbols nach den Regeln der gewählten lexicographischen Darstellung. So ist es z.B. üblich, dass gewisse Symbole aus mehreren Zeichen bestehen. Der Scanner bildet sodann Zeichenfolgen in Symbole ab. Wir erkennen darin den Unterschied zwischen Zeichen und Symbolen: *Zeichen* sind die atomaren Einheiten im Eingabetext des Computers, *Symbole* sind die Grundsteine der durch die Syntax unabhängig von bestimmten Zeichensätzen definierten Sprache.

Bevor wir zur Konstruktion eines Parsers schreiten, ist zu verfizieren, ob die vorliegende Syntax den einschränkenden Regeln 1 und 2 genügt. Danach wenden wir "mechanisch" die Uebertragungsregeln B1 - B6 an und erhalten dabei das folgende *Programm 2.*

```
program EBNF(input,output);
label 99;
const empty = " ";
var sym: char;

procedure getsym;
begin
  repeat read(sym); write(sym) until sym <> " "
end (*getsym*) ;

procedure error;
begin writeln;
  writeln(" Syntax not in correct EBNF"); goto 99
end (*error*) ;

procedure expression;

   procedure term;

      procedure factor;
      begin if sym in ["a".."z"] then
         begin (*nonterminal symbol*)  getsym
         end else
         if sym = """" then
         begin (*terminal symbol*) getsym; getsym;
            if sym <> """" then error else getsym;
```

```
        end else
        if sym = "(" then
        begin getsym; expression;
           if sym = ")" then getsym else error
        end else
        if sym = "[" then
        begin getsym; expression;
           if sym = "]" then getsym else error
        end else
        if sym = "{" then
        begin getsym; expression;
           if sym = "}" then getsym else error
        end else error
     end (*factor*) ;

   begin (*term*)  factor;
      while sym in ["a".."z", """", "(", "[", "{" ] do factor
   end (*term*) ;

begin (*expression*)  term;
   while sym = "|" do
      begin getsym; term
      end
end (*expression*) ;

begin (*main program*)
   while not eof(input) do
      begin (*read a production*)  write(" "); getsym;
         if sym in ["a" .. "z"] then getsym else error;
         if sym = "=" then getsym else error;
         expression; if sym <> "." then error;
         writeln; readln
      end ;
 99: end .
```

Programm 2: Syntax Analysis für EBNF

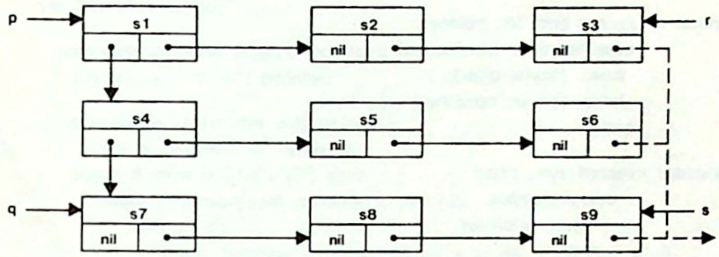
**Schritt 3**

Das nächste Ziel ist die Erweiterung des erhaltenen Programms in einen Uebersetzer, der die gewünschte Datenstruktur erzeugt. Leider lässt sich aber dieser Schritt nicht in einer Art formalisieren oder systematisieren wie der letzte. Mangels formaler Aufbauregeln greifen wir zum naheliegenden Rezept, die jeder syntaktischen Einheit entsprechende Struktur nochmals aufzuzeichnen, und die Parameter der sie erzeugenden Prozedur genau zu definieren. Jede aufgerufene Prozedur erzeugt also eine dem von ihr eingelesenen Text entsprechende Datenstruktur. Diese wird sodann von der aufrufenden Prozedur als Baustein in der von ihr selbst aufgebauten Datenstruktur eingefügt. Dieser Vorgang ist natürlich im allgemeinen rekursiv. Da es sich bei den erzeugten Produkten um durch Zeiger verkettete Strukturen handelt, liegt

es auf der Hand, als Resultate nicht die Strukturen selbst, sondern ihre Zeiger zu verwenden. Jede der drei Generierungs- Prozeduren *factor, term* und *expression* erhält vier Parameter, die mit *p, q, r* und *s* bezeichnet sind. Ihre Bedeutung ist aus der folgenden Datenstruktur ersichtlich, die dem Syntax- Ausdruck

s1 s2 s3 | s4 s5 s6 | s7 s8 s9

entspricht. Die vier Zeiger sind das Resultat der Generierung durch die Prozedur *expression.*




Der Hauptzweck der Prozedur *factor* ist das Erzeugen von neuen Elementen (Knoten) in der Datenstruktur, die den gelesenen Symbolen entsprechen. Die beiden Prozeduren *term* und *expression* hingegen haben die Aufgabe, diese Elemente miteinander zu verketten. Dabei verwendet *term* das Zeigerfeld *suc*, und *expression* benützt das Feld *alt*. Die Einzelheiten sind aus *Programm 3* ersichtlich.

Es sei hier noch auf eine Einzelheit bei der Verarbeitung von nicht-terminalen Symbolen hingewiesen. Möglicherweise kann ein solches Symbol innerhalb eines Faktors vorkommen, bevor es als Linksteil einer Produktion auftritt und dabei definiert wird. Es ist daher notwendig festzuhalten, ob ein vorgekommenes Symbol bereits definiert ist. Dazu werden alle nicht-terminalen Symbole in Records vom Typ *header* registriert, und das Feld *entry* gibt an, ob eine entsprechende Datenstruktur bereits aufgebaut wurde. Diese Records werden in einer Liste linear verkettet. Die Variable *list* stellt den Anfang, *sentinel* das Ende dieser Kette dar, wobei das Ende ein fiktives Element mit dem Zweck ist, den Suchvorgang zu vereinfachen. Dieser Suchvorgang wird von der Hilfsprozedur *find(sym,h)* verkörpert. Wird das Symbol *sym* gefunden, so wird der Zeiger zum entsprechenden Element der Variablen *h* zugewiesen. Ist das Symbol nicht in der Liste vorhanden, so wird ein Element angfügt. Weitere Einzelheiten sind dem ersten Teil von *Programm 3* zu entnehmen.

Schritt 4

Der Uebersetzer wird nun mit dem allgemeinen Parser aus Abschnitt 5 vereinigt. Sowohl das Programm wie die Eingabedaten bestehen aus drei Teilen. Zuerst folgt der Parser für EBNF, respektive die Syntax der zu behandelnden Sprache ausgedrückt in EBNF Produktionen. Dann folgt eine Anweisung, die das Startsymbol einliest. Schliesslich folgen der allgemeine Parser, beziehungsweise die der vorher definierten

Sprache angehörigen und zu erkennenden Sätze. (Das Programm erwartet, dass das Startsymbol durch ein vorangehendes S-Zeichen gekennzeichnet wird.)

```
program Generalparser(input,output);
label 99;
const empty = " ";
type pointer = ↑node;
   hpointer = ↑header;

   node =  record suc, alt: pointer;
           case terminal: boolean of
            true:  (tsym: char);
            false: (nsym: hpointer)
           end ;

   header = record sym: char;
                entry: pointer;
                    suc: hpointer
                end ;

var list, sentinel, h: hpointer;
    q,r,s: pointer;
    sym: char;
    ok: boolean;

procedure getsym;
begin
   repeat read(sym); write(sym) until sym <> " "
end (*getsym*) ;

procedure find(s: char; var h: hpointer);
(*locate nonterminal symbol s in list, if not present, insert it*)
   var h1: hpointer;
begin h1 := list; sentinel↑.sym := s;
   while h1↑.sym <> s do h1 := h1↑.suc;
   if h1 = sentinel then
      begin (*insert*) new(sentinel);
         h1↑.suc := sentinel; h1↑.entry := nil
      end ;
   h := h1
end (*find*) ;

procedure error;
begin writeln;
   writeln(" Syntax not in correct EBNF"); goto 99
end (*error*) ;
```

```
procedure link(p,q: pointer);
   var t: pointer;
begin (*insert q in places indicated by linked chain p*)
   while p <> nil do
      begin t := p; p := t↑.suc; t↑.suc := q
      end
end (*link*) ;

procedure expression(var p,q,r,s: pointer);
   var q1, s1: pointer;

    procedure term(var p,q,r,s: pointer);
       var p1, q1, r1, s1: pointer;

       procedure factor(var p,q: pointer);
          var a: pointer;  h: hpointer;
       begin if sym in ["a".."z"] then
          begin (*nonterminal symbol*)  new(a);  find(sym,h);
             with a↑ do
                begin terminal := false; nsym := h; alt := nil; suc := nil
               end ;
             p := a; q := a; r := a; s := a; getsym
          end else
          if sym = """" then
          begin (*terminal symbol*) getsym; new(a);
             with a↑ do
                begin terminal := true; tsym := sym; alt := nil; suc := nil
                end ;
             p := a; q := a; r := a; s := a; getsym;
             if sym = """" then getsym else error
          end else
          if sym = "(" then
          begin getsym; expression(p,q,r,s);
             if sym = ")" then getsym else error
          end else
          if sym = "[" then
          begin getsym; expression(p,q,r,s); new(a);
             with a↑ do
                begin terminal := true; tsym := empty; alt := nil; suc := nil
                end ;
             q↑.alt := a; s↑.suc := a; q := a; s := a;
             if sym = "]" then getsym else error
          end else
          if sym = "{" then
          begin getsym; expression(p,q,r,s); link(r,p); new(a);
             with a↑ do
                begin terminal := true; tsym := empty; alt := nil; suc := nil
```

```
                end ;
            q↑.alt := a; q := a; r := a; s := a;
            if sym = "}" then getsym else error
          end else error
        end (*factor*) ;

    begin (*term*)  factor(p,q,r,s);
        while sym in ["a"..."z", """", "(", "[", "{" ] do
            begin factor(p1,q1,r1,s1);  link(r,p1); r := r1; s := s1
            end
    end (*term*) ;

begin (*expression*)  term(p,q,r,s);
   while sym = "|" do
      begin getsym; term(q↑.alt, q1, s↑.suc, s1); q := q1; s := s1
      end
end (*expression*) ;

procedure parse(goal: hpointer; var match: boolean);
   var s: pointer;
begin s := goal↑.entry;
   repeat
      if s↑.terminal then
      begin if s↑.tsym = sym then
                begin match := true; getsym
                end
            else match := (s↑.tsym = empty)
      end
      else parse(s↑.nsym, match);
      if match then s := s↑.suc else s := s↑.alt
   until s = nil
end (*parse*) ;

begin  (*main program, read productions*)
   getsym; new(sentinel); list := sentinel;
   while sym <> "$" do
   begin find(sym,h);
      getsym; if sym = "=" then getsym else error;
      expression(h↑.entry, q,r,s); link(r, nil);
      if sym <> "." then error;
      writeln; readln; getsym
   end ;

   h := list; ok := true;  (*check whether all symbols are defined*)
   while h <> sentinel do
   begin if h↑.entry = nil then
         begin writeln(" undefined symbol ", h↑.sym); ok := false
         end ;
```

```
      h := h↑.suc
    end ;
    if not ok then goto 99;

  (*read goal symbol*)
    getsym; find(sym,h); readln; writeln;

  (*read and parse sentences*)
    while not eof(input) do
      begin write(" "); getsym; parse(h,ok);
        if ok and (sym=".") then writeln("  correct")
                            else writeln("  incorrect");
        readln
      end ;
99: end .
```

Programm 3: Allgemeiner Parser

Es ist bemerkenswert, dass dieses Programm einzig durch Einfügen von zusätzlichen Anweisungen in ein bestehendes Programm erzeugt wurde. Das bestehende Programm befasste sich lediglich mit der Erkennung von Sätzen, und kann als Rahmen für ein neues Programm betrachtet werden, das Sätze nicht nur erkennt, sondern in Datenstrukturen übersetzt. Diese Methode der Programm- Entwicklung ist sehr empfehlenswert. Sie gestattet nämlich die Konzentration auf einen bestimmten Aspekt (z.B. die Satzerkennung), bevor andere Aspekte (z.B. die Erzeugung von Datenstrukturen) ebenfalls berücksichtigt werden müssen. Dadurch wird die Arbeit des systematischen Aufbaus und der Verifikation bedeutend erleichtert. Zumindest hilft die Methode dem Programmierer, sich besser und vor allem fortlaufend von der Richtigkeit des komplexen Programmes überzeugen zu können. Die Methode wird *schrittweise Verfeinerung* (stepwise refinement) oder *schrittweiser Ausbau* genannt. In unserem eher einfachen Beispiel dieser Technik besteht der Ausbau lediglich aus zwei Schritten. Kompliziertere Sprachen und komplexere Uebersetzungsaufgaben verlangen oft eine wesentlich grössere Zahl solcher Schritte. Eine sehr ähnliche Entwicklung in drei Schritten wird in Kapiteln 8 bis 11 behandelt.

*Programm 3* zeigt deutlich, dass der Tabellen-gesteuerte - oder eher Datenstruktur-gesteuerte - Algorithmus der Satzzerlegung einen Grad von Flexibilität und Einfachheit aufweist, wie er dem direkt programmierten Parser fehlt. Diese zusätzliche Flexibilität ist, obwohl sie im allgemeinen nicht benötigt wird, der eigentliche Kern der Idee der erweiterbaren Sprachen (*extensible languages*). Dies sind Sprachen, beziehungsweise Uebersetzer, die sich durch Hinzufügen von weiteren Syntaxregeln leicht erweitern lassen. Der Programmierer kann also nach Belieben für jedes Programm Spracherweiterungen angeben, und zwar in Form von Produktionen, welche die Uebersetzer-Tabellen vorübergehend vergrössern. Ein noch ehrgeizigerer Plan lässt es sogar zu, dass die Sprache für lokale Bereiche innerhalb eines Programmes beliebig erweiterbar (und einschränkbar) ist. Dies wird möglich, wenn

*Programm 3* derart abgeändert wird, dass sich Abschnitte von Produktionen und Sätzen (Programmteilen) in der Eingabe abwechseln lassen. Ob sich eine solche Flexibilität in der Praxis als wünschbar oder sogar nützlich erweist, ist allerdings mehr als fraglich. Entwicklungen von solchen erweiterbaren Sprachen und deren Uebersetzer sind denn auch bisher von wenig Erfolg belohnt worden. Der Hauptgrund liegt wohl darin, dass es ganz wesentlich schwieriger ist, die *Bedeutung* der angefügten Erweiterungen zu erklären als deren Struktur. Dieser Teil der Aufgabe hat bisher jedem Versuch einer leicht verständlichen Formalisierung widerstanden. Abgesehen davon liegt der Zweck von Sprachen in der gegenseitigen Verständigung. Dazu sind immer ein Konsensus und Konventionen notwendig. Die beliebige Erfindbarkeit von eigenen Sprachen läuft aber diesem Endziel offensichtlich zuwider. Wir ziehen daraus die Konsequenzen und konzentrieren uns im Rest dieses ~~Kapitels~~ ausschliesslich auf die Entwicklung eines einfachen Uebersetzers für eine einzige, vorgegebene, kleine Programmiersprache.

# 7. Die Programmiersprache PL/0.

Dieses und die folgenden Kapitel sind der Entwicklung einer elementaren Programmiersprache und ihres Compilers gewidmet. Sie heisst PL/0. Einerseits ist es unumgänglich, diese Sprache und den Compiler einfach zu halten, damit sie in den Rahmen dieses Buches passen. Anderseits liegt der Wunsch nahe, die wichtigsten Grundprinzipien von Programmiersprachen und Compilations-Techniken darlegen zu können. Daraus ergeben sich bereits die Randbedingungen für die Art der zu wählenden Sprache PL/0. Ohne Zweifel könnte sie auch einfacher oder komlizierter gewählt werden. PL/0 stellt lediglich einen der möglichen Kompromisse zwischen genügender Einfachheit für eine klare Darstellung und genügender Komplexität dar, die das ganze Projekt realistisch und lohnend macht.

Was Programmstrukturen betrifft, so ist PL/0 verhältnismässig gut ausgebaut. Als elementare Anweisung finden wir die Wertzuweisung. Die zusammengesetzten Anweisungen verkörpern die Konzepte der Anweisungsfolge, der bedingten und der repetierten Anweisung, letztere in der Form von konventionellen **if** und **while** Strukturen. PL/0 enthält ferner das wichtige Konzept des Unterprogramms, dargestellt durch Prozedur-Vereinbarungen und Aufrufe.

Im Bereich der Datentypen und Datenstrukturen hingegen ist PL/0 äusserst frugal. Der einzige Typ ist die ganze Zahl (integer), und Datenstrukturen fehlen sogar gänzlich. Es ist also möglich, ganzzahlige Konstanten und Variablen zu vereinbaren, sowie Ausdrücke mit den arithmetischen Grundoperationen zu bilden.

Prozeduren stellen logische Einheiten von Operationen dar. Es liegt daher nahe, das wichtige Konzept der Lokalität von Objekten (Konstanten, Variablen, Prozeduren) zu verbinden. PL/0 bietet die Möglichkeit, Objekte als lokal zu vereinbaren, d.h. so, dass sie nur innerhalb der Prozedur Gültigkeit haben, in der sie vereinbart sind.

Die obige, kurze Charakterisierung der Sprache hat den Hauptzweck, die nötige Intuition zu vermitteln, um die nachfolgend definierte Syntax zu erfassen und den mit den einzelnen Strukturen verknüpften Sinn zu begreifen. Die Syntax ist am Ende dieses Kapitels sowohl in EBNF als auch mittels Graphen dargestellt.

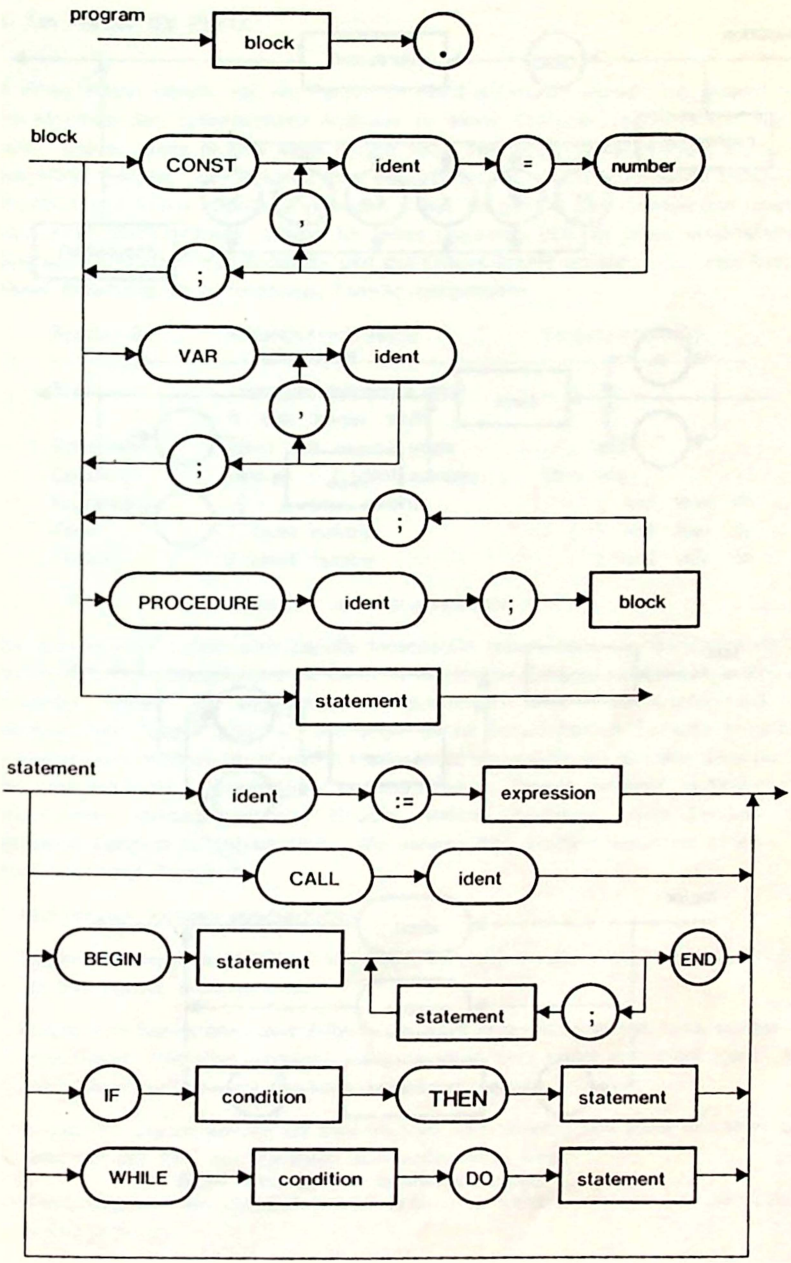
Ein einziges Beispiel eines PL/0 Programmes möge genügen, um die Elemente der Sprache vorzuführen und so ein Bild der Sprache selbst zu vermitteln. Das gewählte Programm enthält Prozeduren zur Multiplikation, Division und Berechnung des grössten gemeinsamen Teilers zweier natürlicher Zahlen.

```
const m = 7, n = 85;
var x,y,z,q,r;

procedure multiply;
   var a,b;
begin a := x; b := y; z := 0;
   while b > 0 do
```

```
        begin if odd b then z := z + a;
              a := 2*a; b := b/2
        end
    end ;

    procedure divide;
        var w;
    begin r := x; q := 0; w := y;
        while w <= r do  w := 2*w;
        while w > y do
        begin q := 2*q; w := w/2;
            if w <= r then
                begin r := r - w; q := q + 1
                end
        end
    end ;

    procedure gcd;
        var f,g;
    begin f := x; g := y;
        while f # g do
            begin if f < g then g := g - f;
                  if g < f then f := f - g;
            end ;
        z := f
    end ;

    begin x := m; y := n; call multiply;
        x := 25; y := 3; call divide;
        x := 84; y := 36; call gcd;
    end .
```

PL/0 Syntax:

```
program   =  block "." .
block     =  [ "const" ident "=" number {"," ident "=" number} ";"]
             [ "var" ident {"," ident} ";"]
             { "procedure" ident ";" block ";"} statement .
statement = [ ident ":=" expression | "call" ident |
             "begin" statement {";" statement} "end" |
             "if" condition "then" statement |
             "while" condition "do" statement ] .
condition = "odd" expression |
             expression ("="|"#"|"<"|"<="|">"|">=") expression .
expression = ["+"|"-"] term {("+"|"-") term}.
term      =  factor {("*"|"/") factor}.
factor    =  ident | number | "(" expression ")" .
```


program
block
.
block
CONST
ident
=
number
,
;
VAR
ident
,
;
;
PROCEDURE
ident
;
block
statement
statement
ident
:=
expression
CALL
ident
BEGIN
statement
END
statement
;
IF
condition
THEN
statement
WHILE
condition
DO
statement

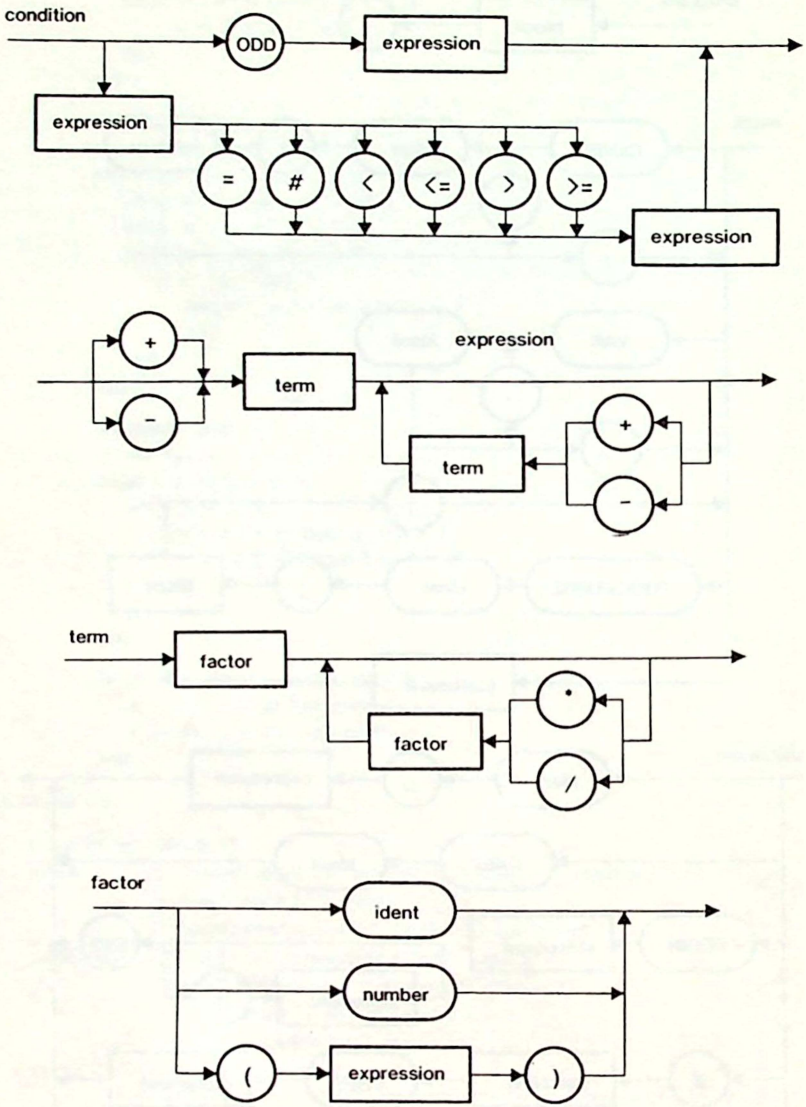



Fig.4. Syntaxdiagramme der Sprache PL/0

## 8. Ein Parser für PL/0.

In einem ersten Schritt soll ein Parser für PL/0 entwickelt werden, der alsdann nach der Methode des schrittweisen Ausbaus zu einem Compiler vervollständigt werden kann. Dieser erste Schritt kann strikte nach den angegebenen Regeln B1 bis B6 ausgeführt werden. Zuvor ist es aber nötig zu prüfen, ob die vorliegende Syntax den Regeln 1 und 2 aus Kapitel 2 genügen. Dies ist anhand der vorliegenden Graphen sehr leicht durchzuführen, sobald für jedes Diagramm (d.h. für jedes nicht-terminale Symbol) die Mengen der Anfangs- und der Folgesymbole ermittelt sind. Das Resultat dieser Ermittlung ist in folgender Tabelle festgehalten.

| Symbol S | Anfangssymbole(S) | Folgesymbole(S) |
|---|---|---|
| Block | const var procedure ident<br>if call begin while | . ; |
| Statement | ident call begin if while | . ; end |
| Condition | odd + - ( ident number | then do |
| Expression | + - ( ident number | . ; ) end then do = . . . |
| Term | ( ident number | . ; ) end then do = + . . . |
| Factor | ( ident number | . ; ) end then do = + * . . . |

Anfangs- und Folgesymbole in PL/0

Der aufmerksame Leser wird bereits festgestellt haben, dass die Grundsymbole der Sprache PL/0 im Unterschied zu früheren Beispielen nicht nur aus einzelnen Zeichen bestehen, sodern im allgemeinen Zeichen*folgen* sind. So werden z.B. die Zeichenfolgen "begin" und ":=", und sogar ganze Bezeichner als Symbole aufgefasst. Nun wird wie schon im Programm 3 ein Scanner eingesetzt, um einzelne Symbole aus der einzulesenden Zeichenfolge herauszulesen. Dieser Scanner verkörpert die sogenannten lexicographischen Regeln, welche bestimmen, wie Symbole aus einzelnen Zeichen aufgebaut sind. Wir nennen den Scanner wiederum *getsym*. Er dient folgenden Zwecken:

1. Leerzeichen werden übersprungen.

2. Sogenannte *reservierte Wörter* wie begin, end, etc. werden erkannt und als Symbole an den Parser weitergeleitet.

3. Folgen von Buchstaben und Ziffern, die nicht reservierte Wörter sind, werden als Bezeichner (identifier) erkannt. Die Variable *sym* erhält den Wert *ident*. Der Bezeichner selbst wird der Variablen *id* zugewiesen.

4. Folgen von Ziffern werden als Zahl erkannt. Die Variable *sym* erhält den Wert *num*, während die Zahl der Variablen *num* zugewiesen wird.

5. Kombinationen von Spezialzeichen (z.B. := ) werden erkannt und als Symbol weitergegeben.

Eine weitere klassische Aufgabe für einen Scanner ist das Ueberspringen von Kommentaren. Dies wird jedoch im PL/0 Compiler nicht gezeigt. Der Scanner (*getsym*) benützt seinerseits eine weitere Hilfsprozedur: *getch* liest bei jedem Aufruf das nächste Zeichen ein. Daneben hat *getch* folgende Aufgaben zu versehen:

1. Es werden Zeilenenden erkannt und als ein Leerzeichen weitergegeben.
2. Der Quelltext wird auf das File *output* kopiert. Damit wird eine Programmliste erstellt.
3. Am Anfang jeder Zeile wird eine Zeilennummer bzw. eine Programmkoordinate ausgegeben.

Der Scanner erfüllt die Aufgabe, genau *ein* Symbol im Text vorauszublicken. Die Entscheidungen des Parsers beruhen auf diesem nächsten Symbol, dargestellt durch die Variable *sym*. Zusätzlich blickt nun aber die Prozedur *getch* noch um ein weiteres Zeichen voraus. Dies ist hier notwendig, weil Symbole aus mehreren Zeichen bestehen können. Die Einzelheiten dieser Routinen sind aus dem Programm 4 ersichtlich, welches einen vollständigen Parser für PL/0 darstellt.

In der Tat ist dieser Parser sogar bereits erweitert worden. Er trägt gelesene Bezeichner in Tabellen ein, wenn sie in Vereinbarungen auftreten. Ihr Vorkommen in Anweisungen bewirkt sodann ein Suchen des Bezeichners in der Tabelle. Damit wird festgestellt, ob er ordnungsgemäss im Programm vereinbart ist. Das Fehlen einer Vereinbarung kann tatsächlich als eine Art syntaktischer Fehler betrachtet werden, da ein nicht vereinbarter Bezeichner als nicht zum Vokabular der Sprache gehörendes Symbol gilt. Der Umstand, dass dieser formale Fehler nur entdeckt werden kann, indem Information in Tabellen festgehalten wird, ist eine Manifestation der Tatsache, dass übliche Programmier- Sprachen nur in beschränktem Sinn kontext-frei sind (siehe Kapitel 1). Bezeichner enthalten eine Bedeutung, die nicht aus der Struktur des Textes allein, sondern vor allem aus ihrer Vereinbarung hervorgeht. Dennoch ist die kontext-freie Syntax ein äusserst nützliches Modell für diese Art von Sprachen. Sie ist durchaus geeignet, um einen einfachen, systematischen Aufbau von Uebersetzern zu ermöglichen. Die so erhaltenen Grundstrukturen der Compiler können dann leicht erweitert werden, um die beschränkte Art der Kontext-Abhängigkeit zu verkraften. Die Tabelle der Bezeichner ist das einzige Instrument, das dazu nötig ist. Ihre Funktion ist leicht verständlich und kann durch wenige, isolierte Prozeduren ausgedrückt werden. Man beachte, dass im vorliegenden Fall die Tabelle als ein Stapel (stack) organisiert ist. Damit wird in einfacher Weise den Regeln der Lokalität entsprochen. Beim Durchsuchen werden lokale Bezeichner zuerst gefunden. Am Ende einer Prozedurvereinbarung, welche einen Gültigkeitsbereich für Bezeichner darstellt, werden die lokalen Bezeichner leicht entfernt, da sie im Stapel zuoberst liegen.

Bevor die einzelnen Prozeduren des Parsers aufgebaut werden, ist es nützlich festzustellen, in welcher Weise sie aufeinander Bezug nehmen. Wir konstruieren dazu

ein sogenanntes *Bezugsdiagramm.* Jeder Graph G wird darin als Knoten dargestellt, von dem ein Verweis (Zeiger) zu jedem Graphen $G_1$ ... $G_n$ ausgeht, auf die in der Definition von G Bezug genommen wird. Dementsprechend zeigt das Diagramm, welche Prozeduren $G_1$ ... $G_n$ von einer gegebenen Prozedur G aus aufgerufen werden.

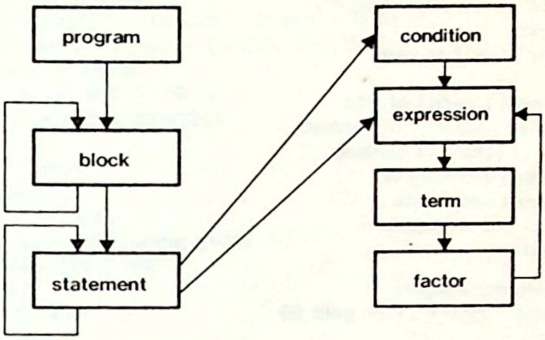



Fig. 5. Bezugsdiagramm der Parser-Prozeduren für PL/0

Jede Schleife im Bezugsdiagramm entspricht einer Rekursion. Es ist daher unerlässlich, dass für die Programmierung eine Sprache zur Verfügung steht, die rekursive Aufrufe von Prozeduren zulässt. Im weiteren zeigt das Bezugsdiagramm, auf welche Art die einzelnen Prozeduren lokal (verschachtelt) vereinbart werden können. Als einzige Struktur, auf die nie Bezug genommen wird, erscheint das Programm. Die entsprechende Prozedur kann daher ebensogut als Hauptprogramm des Compilers gelten. Ausser ihr und der Prozedur *block* selbst nimmt keine andere Prozedur Bezug auf *block*: alle weiteren Prozeduren können daher lokal zu *block* vereinbart werden. Selbstverständlich benützen alle Prozeduren die Hilfsprozeduren *getsym* und letztlich auch *getch.*

```
program PL0(input,output);
(*PL/0 compiler, syntax analysis only*)
label 99;

const norw = 11;          (*no. of reserved words*)
   txmax = 100;           (*length of identifier table*)
   nmax = 14;             (*max. no of digits in numbers*)
   al = 10;             (*length of identifiers*)
   chsetsize = 128;     (*for ASCII character set*)

type symbol =
   (nul,ident,number,plus,minus,times,slash,oddsym,
    eql,neq,lss,leq,gtr,geq,lparen,rparen,comma,semicolon,
    period,becomes,beginsym,endsym,ifsym,thensym,
    whilesym,dosym,callsym,constsym,varsym,procsym);
   alfa = packed array [1..al] of char;
   object = (constant,variable,prozedure);
```

```
var ch: char;            (*last character read*)
   sym: symbol;            (*last symbol read*)
   id: alfa;            (*last identifier read*)
   num: integer;          (*last number read*)
   cc: integer;          (*character count*)
   ll: integer;          (*line length*)
   kk: integer;
   line: array [1..81] of char;
   a: alfa;
   word: array [1..norw] of alfa;
   wsym: array [1..norw] of symbol;
   ssym: array [char] of symbol;
   table: array [0..txmax] of
          record name: alfa;
                 kind: object
          end ;

procedure error(n: integer);
begin writeln(" ":cc, "↑", n:2); goto 99
end (*error*) ;

procedure getsym;
   var i,j,k: integer;

   procedure getch;
   begin if cc = ll then
      begin if eof(input) then
              begin write(" program incomplete"); goto 99
              end ;
         ll := 0; cc := 0; write(" ");
         while not eoln(input) do
            begin ll := ll+1; read(ch); write(ch); line[ll] := ch
            end ;
         writeln; ll := ll+1; read(line[ll])
      end ;
      cc := cc+1; ch := line[cc]
   end (*getch*) ;

begin (*getsym*)
   while ch = " " do getch;
   if ch in ["a"..."z"] then
   begin (*identifier or reserved word*)  k := 0;
      repeat if k < al then
         begin k := k+1; a[k] := ch
         end ;
         getch
      until not (ch in ["a"..."z","0"..."9"]);
      if k >= kk then kk := k else
         repeat a[kk] := " "; kk := kk-1
         until kk = k;
      id := a; i := 1; j := norw;
      repeat k := (i+j) div 2;
```

```
        if id <= word[k] then j := k-1;
        if id >= word[k] then i := k+1
      until i > j;
      if i-1 > j then sym := wsym[k] else sym := ident
    end else
    if ch in ["0".."9"] then
    begin (*number*) k := 0; num := 0; sym := number;
      repeat num := 10*num + (ord(ch)-ord("0"));
        k := k+1; getch
      until not (ch in ["0".."9"]);
      if k > nmax then error(30)
    end else
    if ch = ":" then
    begin getch;
      if ch = "=" then
      begin sym := becomes; getch
      end else sym := nul;
    end else
    if ch = "<" then
    begin getch;
      if ch = "=" then
       begin sym := leq; getch
      end else sym := lss
    end else
    if ch = ">" then
    begin getch;
      if ch = "=" then
       begin sym := geq; getch
      end else sym := gtr
    end else
    begin sym := ssym[ch]; getch
    end
end (*getsym*) ;

procedure block(tx: integer);

    procedure enter(k: object);
    begin (*enter object into table*)
      tx := tx + 1;
      with table[tx] do
      begin name := id; kind := k;
      end
    end (*enter*) ;

    function position(id: alfa): integer;
      var i: integer;
    begin (*find identifier id in table*)
      table[0].name := id;  i := tx;
      while table[i].name <> id do  i := i-1;
      position := i
```

```
end (*position*) ;

procedure constdeclaration;
begin if sym = ident then
   begin getsym;
      if sym = eql then
      begin getsym;
         if sym = number then
            begin enter(constant); getsym
            end
         else error(2)
      end else error(3)
   end else error(4)
end (*constdeclaration*) ;

procedure vardeclaration;
begin if sym = ident then
      begin enter(variable); getsym
      end else error(4)
end (*vardeclaration*) ;

procedure statement;
   var i: integer;

   procedure expression;

      procedure term;

         procedure factor;
            var i: integer;
         begin
            if sym = ident then
            begin i := position(id);
               if i = 0 then error(11) else
               if table[i].kind = prozedure then error(21);
               getsym
            end else
            if sym = number then
            begin getsym
            end else
            if sym = lparen then
            begin getsym; expression;
               if sym = rparen then getsym else error(22)
            end
            else error(23)
         end (*factor*) ;

      begin (*term*) factor;
         while sym in [times,slash] do
            begin getsym; factor
            end
      end (*term*) ;
```

```
  begin (*expression*)
    if sym in [plus,minus] then
      begin getsym; term
      end else term;
    while sym in [plus,minus] do
      begin getsym; term
      end
  end (*expression*) ;

  procedure condition;
  begin
    if sym = oddsym then
    begin getsym; expression
    end else
    begin expression;
      if not (sym in [eql,neq,lss,leq,gtr,geq]) then
        error(20) else
      begin getsym; expression
      end
    end
  end (*condition*) ;

begin (*statement*)
  if sym = ident then
  begin i := position(id);
    if i = 0 then error(11) else
    if table[i].kind <> variable then error(12);
    getsym; if sym = becomes then getsym else error(13);
    expression
  end else
  if sym = callsym then
  begin getsym;
    if sym <> ident then error(14) else
      begin i := position(id);
        if i = 0 then error(11) else
        if table[i].kind <> prozedure then error(15);
        getsym
      end
  end else
  if sym = ifsym then
  begin getsym; condition;
    if sym = thensym then getsym else error(16);
    statement;
  end else
  if sym = beginsym then
  begin getsym; statement;
    while sym = semicolon do
      begin getsym; statement
      end ;
    if sym = endsym then getsym else error(17)
  end else
```

```
      if sym = whilesym then
      begin getsym; condition;
         if sym = dosym then getsym else error(18);
         statement
      end
   end (*statement*) ;


begin (*block*)
   if sym = constsym then
   begin getsym; constdeclaration;
      while sym = comma do
         begin getsym; constdeclaration
         end ;
      if sym = semicolon then getsym else error(5)
   end ;
   if sym = varsym then
   begin getsym; vardeclaration;
      while sym = comma do
         begin getsym; vardeclaration
         end ;
      if sym = semicolon then getsym else error(5)
   end ;
   while sym = procsym do
   begin getsym;
      if sym = ident then
         begin enter(prozedure); getsym
         end
      else error(4);
      if sym = semicolon then getsym else error(5);
      block(tx);
      if sym = semicolon then getsym else error(5);
   end ;
   statement
end (*block*) ;

begin (*main program*)
   for ch := chr(0) to chr(chsetsize-1) do ssym[ch] := nul;
   word[ 1] := "begin     "; word[ 2] := "call      ";
   word[ 3] := "const     "; word[ 4] := "do        ";
   word[ 5] := "end       "; word[ 6] := "if        ";
   word[ 7] := "odd       "; word[ 8] := "procedure ";
   word[ 9] := "then      "; word[10] := "var       ";
   word[11] := "while     ";
   wsym[ 1] := beginsym; wsym[ 2] := callsym;
   wsym[ 3] := constsym; wsym[ 4] := dosym;
   wsym[ 5] := endsym;   wsym[ 6] := ifsym;
   wsym[ 7] := oddsym;   wsym[ 8] := procsym;
   wsym[ 9] := thensym;  wsym[10] := varsym;
   wsym[11] := whilesym;
   ssym["+"] := plus;    ssym["-"] := minus;
```

```
  ssym["*"] := times;   ssym["/"] := slash;
  ssym["("] := lparen;  ssym[")"] := rparen;
  ssym["="] := eql;     ssym[","] := comma;
  ssym["."] := period;  ssym["#"] := neq;
  ssym["<"] := lss;     ssym[">"] := gtr;
  ssym[";"] := semicolon;
  page(output);
  cc := 0; ll := 0; ch := " "; kk := al; getsym;
  block(0);
  if sym <> period then error(9);
99: writeln
end .
```

Programm 4: PL/0 Parser

## 9. Die Behandlung von syntaktischen Fehlern.

Bisher erfüllte der Parser nur die eher bescheidene Aufgabe zu bestimmen, ob eine Folge von Symbolen einen syntaktisch korrekten Satz darstelle. Fast als Nebenprodukt entdeckt dabei der Parser die Struktur des gelesenen Textes. Sobald jedoch ein inkorrektes Symbol vorliegt, ist des Parsers Aufgabe erfüllt, und der Vorgang der Satzzerlegung kann abgebrochen werden. Für praktische Zwecke ist dieses Vorgehen natürlich unzulässig. Vielmehr muss ein Compiler in dieser Lage eine entsprechende Fehleranzeige ausgeben und danach den Zerlegungsprozess fortsetzen. Dabei ist es sogar wahrscheinlich, dass weitere Fehler entdeckt werden. Eine Fortsetzung des Prozesses ist aber nur möglich, wenn gewisse Annahmen oder Hypothesen über die Art des Fehlers gemacht werden. Je nach dieser Annahme muss ein gewisser Teil des Textes übersprungen oder eingefügt werden. Solche Annahmen sind notwendig, selbst wenn gar nie die Absicht gehegt wird, das fehlerhafte Programm zu korrigieren und "dennoch" auszuführen. Ohne eine einigermassen zutreffende Hypothese ist eine Fortsetzung der Satzzerlegung schlechthin unmöglich.

Die Technik der Wahl guter Hypothesen ist recht kompliziert. Sie beruht eher auf Heuristiken, da es bisher kaum gelungen ist, das Problem befriedigend zu formalisieren. Der Hauptgrund dabei ist, dass die rein formale Syntax viele Faktoren ausser Acht lässt, die für die Erkennung eines Satzes durch den Menschen wichtig sind. So ist z.B. das Fehlen einer Interpunktion ein sehr häufiger Fehler, und zwar nicht nur in Programmiersprachen. Hingegen ist es eher selten, dass in einem arithmetischen Ausdruck ein Pluszeichen verloren geht. Für den Parser sind jedoch sowohl Semicolon wie Pluszeichen Symbole ohne Klassenunterschied. Dem Programmierer aber erscheint das Semicolon als beinahe redundant, besonders am Zeilenende, während über die Wichtigkeit des Pluszeichens kein Zweifel besteht. Diese Art von Unterschieden muss aber in Betracht gezogen werden, wenn ein System sich in Bezug auf Fehler "vernünftig" verhalten soll. Daraus geht hervor, dass die Fehlerbehandlung entscheidend von der konkreten Sprache abhängt, und daher nur beschränkt durch allgemeingültige Regeln beschrieben werden kann.

Dennoch gibt es einige heuristische Regeln, die über den unmittelbaren Bereich von PL/0 hinaus Gültigkeit zu haben scheinen. Bezeichnenderweise betreffen sie den Entwurf der Sprache und deren Syntax ebensosehr wie die Technik der Fehlerbehandlung selbst. Vor allem steht es ausser jedem Zweifel, dass eine *einfache Sprachstruktur* eine vernünftige Fehlerbehandlung entscheindend erleichtert oder sogar überhaupt erst ermöglicht.

Falls nach der Identifizierung eines Fehlers ein Teil des nachfolgenden Textes übersprungen werden soll, so ist es unerlässlich, dass die Sprache einige *Schlüsselwörter* enthält, deren Missbrauch sehr unwahrscheinlich ist. Sie sind daher zuverlässige Instrumente, um den Parser wieder korrekt zu orientieren. PL/0 macht sich diese Erkenntnis in hohem Masse zu Nutzen, indem jede Anweisung (ausser der

Wertzuweisung) mit einem distinkten Schlüsselwort beginnt: **begin, while, if.** Dasselbe gilt für Vereinbarungen, die mit **const, var,** oder **procedure** beginnen.

Eine zweite Regel betrifft den Aufbau des Parsers in unmittelbarerer Weise. Beim top-down Parsing werden typischerweise Zerlegungsziele in Unterziele zerlegt, die vom Parser rekursiv in Angriff genommen werden. Diese zweite Regel besagt, dass im Fall eines Fehlers der Parser sein Ziel nicht einfach aufgeben und den Fehler an die ihn aufrufende Prozedur mitteilen soll, sondern dass der gegenwärtige Parser den Text selber soweit überspringen oder ergänzen soll, dass der aufrufende Parser die Zerlegung in plausibler Weise fortsetzen kann. Die programmiertechnische Konsequenz dieser Regel ist, dass jede Prozedur stets regulär zu Ende geführt wird, d.h. dass sie keinen sogenannten speziellen Fehlerausgang besitzt.

Eine mögliche Interpretation dieser Regel führt dazu, dass in jedem Fall nach einem Fehler der Eingabetext so weit übersprungen wird, bis ein legales Folgesymbol vorliegt. Dies bedingt, dass jede Parserroutine die Menge der am Ort ihres Aufrufs gültigen Folgesymbole kennt. Im ersten Ausbauschritt des Parsers von PL/0 fügen wir daher jeder Prozedur einen Parameter zu, der die entsprechende Symbolmenge anzeigt. Am Ende jeder Prozedur wird ein Test eingefügt, der feststellt, ob das nächste Symbol in dieser Menge enthalten sei. Der Test kann weggelassen werden, falls diese Bedingung bereits aus der Logik des bestehenden Parsers hervorgeht.

Es wäre jedoch sehr kurzsichtig, den Text unter allen Umständen bis zum nächsten legalen Folgesymbol zu überspringen. So ist es doch auch möglich, dass gerade dieses Folgesymbol irrtümlicherweise fehlt. Man denke an das häufige Beispiel eines fehlenden Strichpunktes! Das Ueberspringen des Textes kann in solchen Fällen unannehmbare Folgen haben. Wir ergänzen daher diese Mengen von Folgesymbolen durch zusätzliche Symbole, die zwar nicht legale Folgesymbole sind, aber auf jeden Fall das ziellose Ueberspringen abbremsen sollen. Es ist unter diesen Umständen angezeigt, diese Symbole nicht mehr Folgesymbole, sondern *Stopsymbole* zu nennen. Es liegt auf der Hand, dass diese zusätzlichen Symbole vor allem die erwähnten Schlüsselsymbole sind, die am Anfang wichtiger Satzkonstruktionen stehen.

Aus Gründen der Flexibilität führen wir eine allgemein verwendbare Hilfsprozedur genannt *test* ein, die in der erwähnten Art und Weise eingesetzt wird. Sie besitzt drei Parameter und ist nachfolgend beschrieben. Der Parameter *s1* bestimmt die legalen Folgesymbole. Ist das nächste Symbol nicht in dieser Menge enthalten, so liegt ein syntaktischer Fehler vor. Der Parameter *s2* gibt die zusätzlichen Stopsymbole an. Die Anwesenheit eines dieser Symbole ist unzweifelhaft als Fehler zu betrachten; dennoch darf dieses Symbol nicht übersprungen werden. Der dritte Parameter *n* gibt die Fehlernummer an, welche diesem Fall zugeordnet ist.

```
procedure test(s1, s2: symset;  n: integer);
begin if not (sym in s1) then
          begin error(n); s1 := s1 + s2;
              while not (sym in s1) do getsym
          end
end
```

(Das Symbol + bezeichnet an dieser Stelle eine Mengenvereinigung). Es zeigt sich, dass diese Prozedur sich auch vorteilhaft am Eingang eines Parsers verwenden lässt um festzustellen, ob das nächste Symbol ein legales Anfangssymbol der zu untersuchenden Struktur sei. Diese Vorkehrung ist an allen Stellen empfohlen, wo ein Parser aufgerufen wird ohne vorangehende Prüfung des ersten Symbols. In diesen Fällen wird für s1 die Menge der Anfangssymbole und für s2 die Menge der Folgesymbole eingesetzt. Die Einzelheiten der Verwendung der Prozedur *test* sind dem Programm 5 zu entnehmen, das den zur Fehlerbehandlung dienenden Ausbau des Programms 4 darstellt. Der Uebersichtlichkeit halber ist das gesamte Programm nochmals gelistet mit Ausnahme von Initialisierungen und des Scanners, die unverändert bleiben.

Das bis hierher entwickelte Schema beruht auf dem Prinizip, fehlerhafte Ausschnitte eines Textes zu überspringen und den Zerlegungsprozess an geeigneter Stelle wieder aufzunehmen. Dies ist aber eine unzweckmässige Strategie in all jenen Fällen, wo der Fehler darin besteht, dass ein einziges Symbol *weggelassen* wurde. Die Erfahrung zeigt, dass dies vor allem bei Symbolen mit rein syntaktischer Funktion vorkommt, bei Symbolen also, die nicht selbst irgend eine Operation spezifizieren. Eine hinreichende Lösung besteht darin, dass in diesen isolierten Fällen die Syntax der Sprache selbst erweitert wird. Dies geschieht so, dass das Fehlen des besagten Symbols gleichsam legalisiert wird, wobei natürlich der Parser beim Traversieren des neuen Pfades eine Fehlermeldung ausgibt. Als Beispiel für eine solche Erweiterung der Syntax sei die Struktur der Anweisungsfolge aufgeführt, die es nun gestattet, das Semicolon wegzulassen.

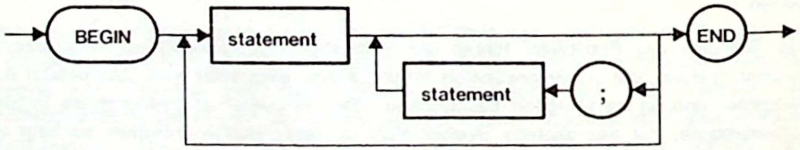



Fig. 6. Modifizierte Syntax der Anweisungsfolge

Die Prozedur, die dieser Struktur zugeordnet ist, lässt sich in üblicher Weise durch die Regeln B1 - B6 herleiten. Im nachfolgenden Programmstück wird allerdings eine Vereinfachung vorgenommen, indem anstelle einer while- eine repeat- Anweisung verwendet wird. Die Menge genannt *statbegsys* bezeichnet die Anfangssymbole der Anweisungsfolge.

```
if sym = beginsym then
begin getsym;
   repeat statement([semicolon, endsym]+fsys);
      while sym = semicolon do
         begin getsym; statement([semicolon, endsym]+fsys)
         end ;
      if sym = endsym then getsym else error( )
   until not (sym in statbegsys)
end
```

Der Erfolgsgrad, mit dem dieses Programm syntaktische Fehler diagnostiziert und überwindet, kann aus dem nachfolgenden kurzen Programmtext abgeschätzt werden. Der Text stellt das in Kapitel 7 angeführte PL/0 Beispiel dar und ist mit Fehlern überladen. Die Fehler werden dennoch vernünftig behandelt, so dass nur selten sogenannte Folgefehler auftreten, die sich beheben, sobald der Originalfehler entfernt wird. Die Tabelle der Fehlermeldungen soll behilflich sein, den vorliegenden Vorgang zu überblicken.

```
program PL0(input,output);
(*PL/0 compiler with syntax error recovery*)
label 99;
const norw = 11;          (*no. of reserved words*)
   txmax = 100;           (*length of identifier table*)
   nmax = 14;             (*max. no of digits in numbers*)
   al = 10;              (*length of identifiers*)
   chsetsize = 128;     (*for ASCII character set*)
   maxerr = 30;           (*max. no. of errors*)

type symbol =
   (nul,ident,number,plus,minus,times,slash,oddsym,
    eql,neq,lss,leq,gtr,geq,lparen,rparen,comma,semicolon,
    period,becomes,beginsym,endsym,ifsym,thensym,
    whilesym,dosym,callsym,constsym,varsym,procsym);
   alfa = packed array [1..al] of char;
   object = (constant,variable,prozedure);
   symset = set of symbol;

var ch: char;             (*last character read*)
   sym: symbol;            (*last symbol read*)
   id: alfa;             (*last identifier read*)
   num: integer;           (*last number read*)
   cc: integer;           (*character count*)
   ll: integer;          (*line length*)
   kk, err: integer;
   line: array [1..81] of char;
   a: alfa;
   word: array [1..norw] of alfa;
   wsym: array [1..norw] of symbol;
   ssym: array [char] of symbol;
   declbegsys,  statbegsys, facbegsys: symset;
```

```
  table: array [0..txmax] of
         record name: alfa;
                kind: object
         end ;

procedure error(n: integer);
begin writeln(" ":cc-1, "↑", n:2);
   err := err + 1; if err > maxerr then goto 99
end (*error*) ;

procedure getsym;
begin (*scanner: see Program 4*)
end (*getsym*) ;

procedure test(s1,s2: symset; n: integer);
begin if not (sym in s1) then
      begin error(n); s1 := s1 + s2;
         while not (sym in s1) do getsym
      end
end (*test*) ;


procedure block(tx: integer; fsys: symset);

   procedure enter(k: object);
   begin (*enter object into table*)
      tx := tx + 1;
      with table[tx] do
      begin name := id; kind := k;
      end
   end (*enter*) ;

   function position(id: alfa): integer;
      var i: integer;
   begin (*find identifier id in table*)
      table[0].name := id;  i := tx;
      while table[i].name <> id do  i := i-1;
      position := i
   end (*position*) ;

   procedure constdeclaration;
   begin if sym = ident then
      begin getsym;
         if sym in [eql,becomes] then
         begin if sym = becomes then error(1);
            getsym;
            if sym = number then
               begin enter(constant); getsym
               end
            else error(2)
         end else error(3)
      end else error(4)
   end (*constdeclaration*) ;
```

```
procedure vardeclaration;
begin if sym = ident then
      begin enter(variable); getsym
      end else error(4)
end (*vardeclaration*) ;

procedure statement(fsys: symset);
   var i: integer;

   procedure expression(fsys: symset);

      procedure term(fsys: symset);

         procedure factor(fsys: symset);
            var i: integer;
         begin test(facbegsys,fsys,24);
            while sym in facbegsys do
            begin
               if sym = ident then
               begin i := position(id);
                  if i = 0 then error(11) else
                  if table[i].kind = prozedure then error(21);
                  getsym
               end else
               if sym = number then
               begin getsym;
               end else
               if sym = lparen then
               begin getsym; expression([rparen]+fsys);
                  if sym = rparen then getsym else error(22)
               end ;
               test(fsys,[lparen],23)
            end
         end (*factor*) ;

      begin (*term*) factor(fsys+[times,slash]);
         while sym in [times,slash] do
            begin getsym; factor(fsys+[times,slash])
            end
      end (*term*) ;

   begin (*expression*)
      if sym in [plus,minus] then
         begin getsym; term(fsys+[plus,minus])
         end else term(fsys+[plus,minus]);
      while sym in [plus,minus] do
         begin getsym; term(fsys+[plus,minus])
         end
   end (*expression*) ;

   procedure condition(fsys: symset);
   begin
      if sym = oddsym then
```

```
      begin getsym; expression(fsys);
      end else
      begin expression([eql,neq,lss,gtr,leq,geq]+fsys);
         if not (sym in [eql,neq,lss,leq,gtr,geq]) then
            error(20) else
         begin getsym; expression(fsys)
         end
      end
   end (*condition*) ;

begin (*statement*)
   if sym = ident then
   begin i := position(id);
      if i = 0 then error(11) else
      if table[i].kind <> variable then error(12);
      getsym; if sym = becomes then getsym else error(13);
      expression(fsys);
   end else
   if sym = callsym then
   begin getsym;
      if sym <> ident then error(14) else
         begin i := position(id);
            if i = 0 then error(11) else
            if table[i].kind <> prozedure then error(15);
            getsym
         end
   end else
   if sym = ifsym then
   begin getsym; condition([thensym,dosym]+fsys);
      if sym = thensym then getsym else error(16);
      statement(fsys)
   end else
   if sym = beginsym then
   begin getsym; statement([semicolon,endsym]+fsys);
      while sym in [semicolon]+statbegsys do
      begin
         if sym = semicolon then getsym else error(10);
         statement([semicolon,endsym]+fsys)
      end ;
      if sym = endsym then getsym else error(17)
   end else
   if sym = whilesym then
   begin getsym; condition([dosym]+fsys);
      if sym = dosym then getsym else error(18);
      statement(fsys);
   end ;
   test(fsys,[],19)
end (*statement*) ;
```

```
begin (*block*)
  repeat
    if sym = constsym then
    begin getsym;
      repeat constdeclaration;
        while sym = comma do
          begin getsym; constdeclaration
          end ;
        if sym = semicolon then getsym else error(5)
      until sym <> ident
    end ;
    if sym = varsym then
    begin getsym;
      repeat vardeclaration;
        while sym = comma do
          begin getsym; vardeclaration
          end ;
        if sym = semicolon then getsym else error(5)
      until sym <> ident;
    end ;
    while sym = procsym do
    begin getsym;
      if sym = ident then
        begin enter(prozedure); getsym
        end
      else error(4);
      if sym = semicolon then getsym else error(5);
      block(tx, [semicolon]+fsys);
      if sym = semicolon then
        begin getsym; test(statbegsys+[ident,procsym],fsys,6)
        end
      else error(5)
    end ;
    test(statbegsys+[ident],declbegsys,7)
  until not (sym in declbegsys);
  statement([semicolon,endsym]+fsys);
  test(fsys,[],8);
end (*block*) ;

begin (*main program, for initializations, see Program 4*)
  declbegsys := [constsym,varsym,procsym];
  statbegsys := [beginsym,callsym,ifsym,whilesym];
  facbegsys := [ident,number,lparen];
  page(output); err := 0;
  cc := 0; ll := 0; ch := " "; kk := al; getsym;
  block(0,[period]+declbegsys+statbegsys);
  if sym <> period then error(9);
99: writeln
end .
```

Programm 5: PL/0 Parser mit Fehlerbehandlung

1 Verwende "=" anstatt ":=" !
2 Nach "=" muss eine Zahl folgen.
3 Nach dem Bezeichner muss "=" folgen.
4 Nach "const", "var" oder "procedure" muss ein Bezeichner folgen.
5 Strichpunkt (oder Komma) fehlt.
6 Inkorrektes Symbol nach der Prozedurvereinbarung.
7 Hier wird eine Anweisung erwartet.
8 Inkorrektes Symbol nach dem Anweisungsteil im Block.
9 Hier wird ein Punkt erwartet.
10 Inkorrektes Symbol in einer Anweisung.
11 Dieser Bezeichner ist nicht vereinbart.
12 Zuweisung zu Konstanten und Prozeduren sind nicht erlaubt.
13 Zuweisungsoperator ist ":=".
14 Nach "call" muss ein Bezeichner folgen.
15 Aufruf (call) einer Konstanten oder Variablen ist nicht erlaubt.
16 Hier wird "then" erwartet.
17 Hier wird ";" oder "end" erwartet.
18 Hier wird "do" erwartet.
19 Auf diese Anweisung folgt ein inkorrekt verwendetes Symbol.
20 Hier wird eine Relation erwartet.
21 Ein Ausdruck darf keinen Prozedurbezeichner enthalten.
22 Schliessende Klammer fehlt.
23 Der vorangehende Faktor kann nicht mit diesem Symbol enden.
24 Ein Ausdruck kann nicht mit diesem Symbol beginnen.
30 Zahl ist zu gross.

Fehlermeldungen des PL/0 Compilers nach Programm 5.

Es dürfte aus den vorangegangenen Ausführungen und Beispielen ersichtlich sein, dass es keine perfekte Strategie der Fehlerbehandlung gibt, die alle korrekten Sätze mit hinreichender Effizienz übersetzt und alle inkorrekten Sätze vernünftig diagnostiziert. Jede Strategie wird gewisse abstruse Symbolfolgen in einer Weise behandeln, die ihrem Autor unerwartet erscheinen. Das wesentliche Merkmal eines guten Compilers jedoch ist, dass (1) keine Symbolfolge zum Zusammenbruch des Compilers führen kann, und (2) dass häufige, echte Fehler korrekt diagnostiziert werden und keine zusätzlichen, späteren Fehlermeldungen verursachen. Die hier vorgetragene Strategie arbeitet zufriedenstellend, obwohl es selbstverständlich Möglichkeiten zur Verbesserung gibt. Bemerkenswert ist an dieser Methode, dass der obige Compiler systematisch durch Anwendung einiger weniger Grundregeln entstanden ist. Diese Regeln wurden lediglich durch geschickte Wahl einiger Parameter ergänzt, die auf Erfahrung im Gebrauch der Sprache beruht.

## 10. Ein Interpreter für PL/0.

Es ist in der Tat bemerkenswert, dass ein PL/0 Compiler bis hierher ohne Kenntnis des Computers entwickelt wurde, für den er Programme übersetzen soll. Aber aus welchem Grunde sollte die Struktur der Zielmaschine die Strategie der syntaktischen Analyse und der Fehlerbehandlung beeinflussen? Ganz im Gegenteil muss eine solche Beeinflussung sogar vermieden werden. Anstatt dessen wird eine geeignete Art der Code Generierung für einen beliebigen Computer nach dem Prinzip des schrittweisen Ausbaus auf das bestehende Gerüst des Compilers superponiert. Bevor wir diesen Schritt in Angriff nehmen können, ist es allerdings nötig, dass wir uns auf eine Zielmaschine und damit eine Zielsprache der Uebersetzung festlegen.

Um den Compiler verhältnismässig einfach gestalten zu können und die Entwicklung nicht mit unwichtigen Einzelheiten zu belasten, die lediglich durch bestimmte Eigenschaften des gewählten Computers bedingt sind, postulieren wir einen Computer nach unserer eigenen Wahl. Daraus ergibt sich der beträchtliche Vorteil, dass er auf die Bedürfnisse der Quellsprache PL/0 zugeschnitten werden kann. Dieser Computer existiert zwar nicht als reale Maschine; wir nennen ihn daher *hypothetisch*. Da ein Computer jedoch stets Befehle nach einem bestimmten Rezept (Algorithmus) ausführt, lässt er sich durch ein Programm beschreiben. Da ein existierender Computer sodann benützt werden kann, um dieses Programm zu interpretieren, wird es ein *Interpreter* genannt. Der hypothetische Computer wird also durch den Interpreter simuliert; er besitzt daher mindestens eine semi-reale Existenz.

Es ist nicht das Ziel dieser Ausführungen, auf die Beweggründe einzugehen, die zur Wahl der nachfolgend beschriebenen Computerstruktur und ihren Einzelheiten geführt haben. Vielmehr soll dieses Kapitel als eine Art beschreibendes Manual dienen, das aus einer informalen Einführung und einer formalen Definition des Computers als Programm besteht. Diese Formalisierung mag sogar als Beispiel für eine exakte, algorithmische Beschreibung von Prozessoren aufgefasst werden.

Der PL/0 Computer besteht aus zwei Speichern, einem Befehlsregister und drei Adressregistern. Der Programmspeicher (genannt *code*) wird durch den Compiler geladen und bleibt während der Ausführung eines Programmes unverändert. Der Datenspeicher S ist durchwegs als Stapel (*stack*) verwendet. Alle arithmetischen Operatoren finden ihre Operanden zuoberst auf dem Stapel und ersetzen sie durch das Resultat. Das oberste Element wird durch das Adress- bzw. Indexregister T (*top*) gekennzeichnet. Das Befehlsregister I enthält den gegenwärtig interpretierten Befehl. Der Programmzähler P enthält die Adresse bzw. den Index des nächsten Befehls, der zur Interpretation aus dem Programmspeicher zu holen ist.

Jede Prozedur in PL/0 kann lokale Variablen besitzen. Da Prozeduren rekursiv aufrufbar sind und jede Rekursion eigene lokale Variablen erheischt, ist es unmöglich, lokalen Variablen bereits bei der Compilation feste Speicherplätze zuzuweisen. Anstatt dessen wird während der Ausführung bei jedem Aufruf ein entsprechender

Speicherplatz für den Satz lokaler Variablen festgelegt. Am Ende der Prozedur wird er wieder freigegeben. Weil jede Prozedur Q, die nach einer Prozedur P aufgerufen wurde, beendet wird, bevor P beendet ist (last in first out), ist die Speicherverwaltung nach dem Stapelprinzip die naheliegende Lösung. Beim Aufruf wird der Prozedur zuoberst im Stapel ein Platz zugewiesen; er wird *Prozedur-Segment* oder auch "activation record" genannt. Jede Prozedur speichert darin zuerst einige "private" Daten ab, zumindest die Programmadresse ihres Aufrufs (oft "return address" genannt) und die Segment-Adresse der aufrufenden Prozedur. Diese beiden Daten sind unerlässlich, um nach Beendigung der Prozedur den vor dem Aufruf geltenden Zustand wieder herstellen zu können. Wir fassen sie als implizite lokale Daten einer Prozedur auf. Sie stellen den sogenannten *Segment-Descriptor* dar mit den zwei Komponenten RA (return address) und DL (dynamic link). Die DL-Werte verketten alle Segmente im Stapel; der Anfang dieser Kette befindet sich im *Basis-Adress* Register B (siehe Fig. 7).

Da die eigentliche Speicherplatzzuweisung erst während der Interpretation des compilierten Programmes stattfindet, kann der Compiler keine endgültigen Adressen in die Befehle einbauen. Er kann lediglich den Platz einer Variablen innerhalb eines Segmentes bestimmen. Diese relative Adresse wird *offset* genannt. Der Interpreter muss sodann, um eine absolute Adresse zu erhalten, die Basis-Adresse des betreffenden Segmentes zum offset hinzuzählen. Wenn eine Variable lokal zur gegenwärtig ausgeführten Prozedur ist, so kann diese Basis-Adresse dem Register B entnommen werden. Ansonst muss sie ermittelt werden, indem die Adress-Kette DL bis zum betreffenden Segment durchlaufen wird. Der Compiler kennt jedoch nur die Tiefe der statischen Verschachtelung von Prozeduren, während die Kette DL die dynamische Folge von Aufrufen festhält. Leider sind die beiden Folgen nicht notwendigerweise identisch. Dies wird durch das folgende Beispiel veranschaulicht.

Wir nehmen an, dass eine Prozedur A eine Prozedur B aufruft, die lokal zu A ist. In gleicher Weise ruft B eine Prozedur C, die lokal zu B vereinbart ist. Schliesslich ruft C die Prozedur B (rekursiv). Wir sagen, dass A auf Niveau 1, B auf Niveau 2, und C auf Niveau 3 vereinbart sind. Findet nun in B ein Zugriff zu einer Variablen von A statt, so weiss der Compiler, dass ein Niveau-Unterschied von 1 besteht. Ein Zugriff entlang der dynamischen Kette DL hingegen führt im ersten Schritt in das Segment der Prozedur C ! Aus diesem Beispiel geht hervor, dass eine zweite Verkettung nötig ist, welche den Zugriffspfad zu Variablen korrekt widerspiegelt (siehe Fig. 7). Wir nennen diese zweite Kette SL (static link).

Adressen werden deshalb vom Compiler als Zahlen*paare* dargestellt. Die erste Zahl bezeichnet die Niveau-Differenz zwischen dem Aufrufer und der zugegriffenen Variablen und bestimmt die Anzahl der in der SL-Kette zu durchlaufenden Glieder. Die zweite Zahl ist der *offset*, d.h. die Adresse relativ zum Anfang des die Variable enthaltenden Segmentes.

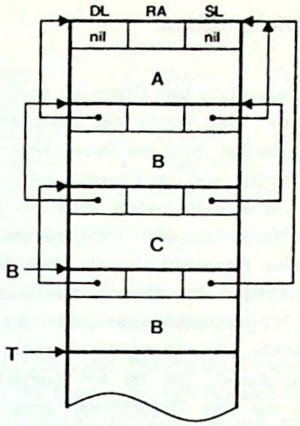



Fig. 7. Stapel und Segment-Verkettung

Der Befehlssatz des PL/0 Computers ist direkt auf die Erfordernisse der Sprache zugeschnitten. Er enthält die folgenden Befehle:

1. Ein Befehl LIT, um Zahlen (literals) in den Stapel zu bringen.
2. Ein Befehl LOD, um Variablenwerte in den Stapel zu laden.
3. Ein Befehl STO, um Werte aus dem Stapel abzuspeichern (store).
4. Ein Befehl CAL, um eine Prozedur aufzurufen (call).
5. Ein Befehl INT, um Speicher zu reservieren (increment T).
6. Befehle JMP und JPC, um im Programm zu springen (jump).
7. Ein Satz von arithmetischen und relationalen Operatoren OPR.

Das Befehlsformat wird durch die Notwendigkeit dreier Befehlskomponenten bestimmt. Jeder Befehl enthält einen Befehlscode f, der die Art des Befehls kennzeichnet, sowie einen (oder zwei) Parameter. Im Fall der Operatoren gibt der Parameter die Identität des Operators an, da der Befehlscode lediglich eine Befehlsklasse bezeichnet. Bei den Befehlen LIT und INT ist der Parameter eine Zahl, bei JMP, JPC und CAL eine Programmadresse, bei LOD und STO eine Datenadresse.

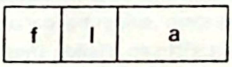



Fig. 8. Befehlsformat des PL/0 Computers

Die Einzelheiten und die genaue Funktionsweise des PL/0 Computers sind dem *Programm 6* zu entnehmen. Es enthält den Interpreter als Prozedur, welche aufgerufen wird, nachdem der Compiler den zu interpretierenden Code erzeugt hat. Das Programm ist ein Beispiel für ein vollständiges, interpretatives System, das eine gegebene Sprache übersetzt und auswertet.

## 11. Die Erzeugung von Befehls-Code.

Um Befehle erzeugen zu können, muss der Compiler sowohl die Art des Befehls wie auch dessen Parameter kennen. Die Werte der Parameter, die sich auf Variablen, Konstanten und Prozeduren beziehen, sind mit Bezeichnern verknüpft. Die Parameter werden während der Uebersetzung von Vereinbarungen ermittelt und in der Tabelle des Compilers eingetragen. Zu diesem Zweck wird die Struktur der Tabelle (siehe *Programm 4*) erweitert, um die relevanten Attribute der Bezeichner festzuhalten. Identifiziert der Bezeichner eine Konstante, so ist sein Attribut eine Zahl; bei einer Variablen ist es eine Datenadresse (bestehend aus einem Zahlenpaar), bei einer Prozedur sind es eine Programmadresse und die Schachtelungstiefe der Vereinbarung. Die entsprechende Erweiterung der Struktur der Tabelle des Compilers ist dem *Programm 6* zu entnehmen. Sie ist ein bemerkenswertes Beispiel eines schrittweisen Ausbaus nicht nur des Programms, sondern auch der zugehörigen Datenstruktur.

Während bei der Verarbeitung von Konstanten-Vereinbarungen der Wert der Konstanten direkt dem Programm entnommen werden kann, müssen bei den Variablen-Vereinbarungen die Adressen vom Compiler berechnet werden. Dieser Prozess ist im allgemeinen nicht immer einfach. PL/0 jedoch ist genügend primitiv, um sequentielle Speicherzuweisung zu ermöglichen. Dies besagt, dass ein Adresszähler, der den Platz einer Variablen bestimmt, bei jeder Vereinbarung einfach um 1 erhöht wird. Dies ist daher zugelassen, weil jede Variable in der PL/0 Maschine genau eine Speicherzelle beansprucht. Der Zähler *dx* muss an jedem Prozeduranfang neu initialisiert werden, da lediglich relative Adressen generiert werden (siehe Kapitel 10). Bei jeder Vereinbarung wird die Prozedur *enter* aufgerufen, welche die notwendigen Eintragungen in der Tabelle vornimmt.

Unter diesen Umständen ist es verhältnismässig einfach, Programme in Befehlsfolgen zu übersetzen. Dank der Organisation des Datenspeichers als Stapel gilt dies ganz besonders für Ausdrücke. Die Hauptaufgabe des Compilers ist das Umordnen von Operanden und Operatoren in die sogenannte *postfix* Notation. Darin *folgt* ein Operator stets seinen Operanden, anstatt wie bei der *infix* Notation zwischen ihnen eingebettet zu sein. Die postfix Form wird oft auch *klammerfrei* genannt, weil sie gestattet, Ausdrücke ohne Klammern eindeutig darzustellen. Einige infix Ausdrücke und ihre entsprechenden postfix Formen stellen diese Eigenschaft unter Beweis:

| | |
|---|---|
| x + y | x y + |
| (x-y) + z | x y - z + |
| x - (y+z) | x y z + - |
| (x+y)/(z-w) | x y + z w - / |

Die Uebersetzungsregeln für die syntaktischen Einheiten des Ausdrucks, Terms und Faktors lassen sich wie folgt schematisieren, wobei $T(\xi)$ die Uebersetzung der Symbolfolge $\xi$ bedeutet:

```
T("+" term)                  =  T(term)
T("-" term)                  =  T(term) "-"
T(term1 "+" term2)           =  T(term1)  T(term2)  "+"
T(term1 "-" term2)           =  T(term1)  T(term2)  "-"
T(factor1 "*" factor2)       =  T(factor1)  T(factor2)  "*"
T(factor1 "/" factor2)       =  T(factor1)  T(factor2)  "/"
T( "(" expression ")" )      =  T(expression)
```

Damit reduziert sich die Aufgabe der Prozeduren zur Behandlung der Ausdrücke auf das Verzögern der Uebermittlung der Operatoren. Der Leser sollte sich an dieser Stelle davon überzeugen, dass die Syntax die üblichen Prioritätsregeln von arithmetischen Operatoren korrekt widerspiegelt.

Die Uebersetzung von bedingten und repetierten Anweisungen ist nur unwesentlich komplizierter. Sie erfordert die Erzeugung von Sprungbefehlen. In gewissen Fällen ist nun die Zieladresse dieser Sprünge zur Zeit der Ausgabe dieser Befehle noch nicht bekannt. Es betrifft dies alle Sprünge nach vorwärts. Beharrt man strikte auf dem Prinzip der sequentiellen Uebersetzung des Textes, dann muss in diesen Fällen der Text zweimal durchlesen werden. Diese weitverbreitete Strategie wird ein Zweiphasen-Compiler genannt. Die zweite Phase der Uebersetzung besteht dann im Einsetzen der nunmehr bekannten Zieladressen. Wir wählen aber hier eine andere Lösung, die darin besteht, die Befehle in einem nachträglich beliebig zugreifbaren Array einzutragen, so dass die Zieladressen eingesetzt werden können, sobald sie bekannt sind. Dieses "Flicken" der unvollständigen Befehle wird im Jargon als *fixup* bezeichnet. Die einzige Aufgabe neben der Ausgabe des Sprunges ist dann das Festhalten seines Index im Befehlsspeicher. Dieser Index gibt dann an, wo der "fixup" vorzunehmen ist. Die Einzelheiten sind aus *Programm 6* ersichtlich, und zwar in den Prozeduren, die die if- und while- Anweisungen behandeln. Wir stellen ihre Aufgabe formal durch folgende zwei Regeln dar:

```
T("if" condition "then" statement)   =      T(condition)
                                            JPC L1
                                            T(statement)
                                     L1:  . . .

T("while" condition "do" statement)  =  L1: T(condition)
                                            JPC L2
                                            T(statement)
                                            JMP L1
                                     L2:  . . .
```

Als Beispiel sei nachfolgend die Befehlsfolge angegeben, die sich aus der Compilation der Prozedur zur Multiplikation zweier Zahlen (siehe Kapitel 7) ergibt. Die Kommentare auf der rechten Seite sind hier lediglich zur Information des Lesers angefügt.

```
2   INT   0,5   allocate space
3   LOD   1,3   x
4   STO   0,3   a
5   LOD   1,4   y
6   STO   0,4   b
7   LIT   0,0   0
8   STO   1,5   z
9   LOD   0,4   b
10  LIT   0,0   0
11  OPR   0,12  >
12  JPC   0,29
13  LOD   0,4   b
14  OPR   0,6   odd
15  JPC   0,20
16  LOD   1,5   z
17  LOD   0,3   a
18  OPR   0,2   +
19  STO   1,5   z
20  LIT   0,2   2
21  LOD   0,3   a
22  OPR   0,4   *
23  STO   0,3   a
24  LOD   0,4   b
25  LIT   0,2   2
26  OPR   0,5   /
27  STO   0,4   b
28  JMP   0,9
29  OPR   0,0   return
```

PL/0 Code für die Multiplikationsprozedur aus Kapitel 7.

```
program PL0(input,output);
(*PL/0 compiler with code generation*)
label 99;
const norw = 11;        (*no. of reserved words*)
  txmax = 100;          (*length of identifier table*)
  nmax = 14;            (*max. no of digits in numbers*)
  al = 10;              (*length of identifiers*)
  chsetsize = 128;      (*for ASCII character set*)
  maxerr = 30;          (*max. no. of errors*)
  amax = 2048;          (*maximaum address*)
  levmax = 3;           (*maximium depth of block nesting*)
  cxmax = 200;          (*size of code array*)

type symbol =
  (nul,ident,number,plus,minus,times,slash,oddsym,
   eql,neq,lss,leq,gtr,geq,lparen,rparen,comma,semicolon,
   period,becomes,beginsym,endsym,ifsym,thensym,
   whilesym,dosym,callsym,constsym,varsym,procsym);
  alfa = packed array [1..al] of char;
  object = (constant,variable,prozedure);
```

```
  symset = set of symbol;
  fct = (lit,opr,lod,sto;cal,int,jmp,jpc);        (*functions*)
  instruction = packed record
                  f: fct;              (*function code*)
                  l: 0 .. levmax;      (*level*)
                  a: 0 .. amax;        (*displacement address*)
                end ;
(*  LIT 0,a        :  load constant a
    OPR 0,a        :  exec operation a
    LOD l,a        :  load variable l,a
    STO l,a        :  store variable l,a
    CAL l,a        :  call procedure a at level l
    INT 0,a        :  increment t-register by a
    JMP 0,a        :  jump to a
    JPC 0,a        :  jump conditional to a  *)

var ch: char;                  (*last character read*)
  sym: symbol;                 (*last symbol read*)
  id: alfa;                    (*last identifier read*)
  num: integer;                (*last number read*)
  cc: integer;                 (*character count*)
  ll: integer;                 (*line length*)
  kk, err: integer;
  cx: integer;                 (*code allocation index*)
  line: array [1..81] of char;
  a: alfa;
  code: array [0..cxmax] of instruction;
  word: array [1..norw] of alfa;
  wsym: array [1..norw] of symbol;
  ssym: array [char] of symbol;
  mnemonic: array [fct] of
              packed array [1..5] of char;
  declbegsys,  statbegsys, facbegsys: symset;
  table: array [0..txmax] of
         record name: alfa;
           case kind: object of
           constant: (val: integer);
           variable,prozedure: (level, adr, size: integer)
         end ;

procedure error(n: integer);
begin writeln(" ****", " ":cc-1, "↑", n:2);
  err := err + 1; if err > maxerr then goto 99
end (*error*) ;

procedure getsym;
begin (*scanner: see Program 4*)
end (*getsym*) ;
```

```
procedure gen(x: fct; y,z: integer);
begin if cx > cxmax then
        begin write(" program too long"); goto 99
        end ;
   with code[cx] do
      begin f := x; l := y; a := z
      end ;
   cx := cx + 1
end (*gen*) ;

procedure test(s1,s2: symset; n: integer);
begin if not (sym in s1) then
      begin error(n); s1 := s1 + s2;
         while not (sym in s1) do getsym
      end
end (*test*) ;


procedure block(lev,tx: integer; fsys: symset);
   var dx: integer;          (*data allocation index*)
      tx0: integer;          (*initial table index*)
      cx0: integer;          (*initial code index*)

   procedure enter(k: object);
   begin (*enter object into table*)
      tx := tx + 1;
      with table[tx] do
      begin name := id; kind := k;
         case k of
         constant: begin if num > amax then
                           begin error(31); num := 0 end ;
                     val := num
                  end ;
         variable: begin level := lev; adr := dx; dx := dx+1;
                  end ;
         prozedure: level := lev
         end
      end
   end (*enter*) ;

   function position(id: alfa): integer;
      var i: integer;
   begin (*find identifier id in table*)
      table[0].name := id;  i := tx;
      while table[i].name <> id do  i := i-1;
      position := i
   end (*position*) ;

   procedure constdeclaration;
   begin if sym = ident then
      begin getsym;
         if sym in [eql,becomes] then
```

```
    begin if sym = becomes then error(1);
       getsym;
       if sym = number then
          begin enter(constant); getsym
          end
       else error(2)
    end else error(3)
  end else error(4)
end (*constdeclaration*) ;

procedure vardeclaration;
begin if sym = ident then
    begin enter(variable); getsym
    end else error(4)
end (*vardeclaration*) ;

procedure listcode;
  var i: integer;
begin (*list code generated for this block*)
  for i := cx0 to cx-1 do
    with code[i] do
      writeln(i, mnemonic[f]:5, l:3, a:5)
end (*listcode*) ;

procedure statement(fsys: symset);
  var i,cx1,cx2: integer;

  procedure expression(fsys: symset);
    var addop: symbol;

    procedure term(fsys: symset);
      var mulop: symbol;

      procedure factor(fsys: symset);
        var i: integer;
      begin test(facbegsys,fsys,24);
        while sym in facbegsys do
        begin
          if sym = ident then
          begin i := position(id);
            if i = 0 then error(11) else
            with table[i] do
            case kind of
              constant: gen(lit,0,val);
              variable: gen(lod,lev-level,adr);
              prozedure: error(21)
            end ;
            getsym
          end else
          if sym = number then
          begin if num > amax then
                begin error(31); num := 0
```

```
                end ;
              gen(lit,0,num); getsym
            end else
            if sym = lparen then
            begin getsym; expression([rparen]+fsys);
              if sym = rparen then getsym else error(22)
            end ;
            test(fsys,[lparen],23)
          end
        end (*factor*) ;

    begin (*term*) factor(fsys+[times,slash]);
        while sym in [times,slash] do
          begin mulop := sym; getsym; factor(fsys+[times,slash]);
            if mulop = times then gen(opr,0,4) else gen(opr,0,5)
          end
    end (*term*) ;

  begin (*expression*)
    if sym in [plus,minus] then
        begin addop := sym; getsym; term(fsys+[plus,minus]);
          if addop = minus then gen(opr,0,1)
        end else term(fsys+[plus,minus]);
    while sym in [plus,minus] do
        begin addop := sym; getsym; term(fsys+[plus,minus]);
          if addop = plus then gen(opr,0,2) else gen(opr,0,3)
        end
  end (*expression*) ;

  procedure condition(fsys: symset);
    var relop: symbol;
  begin
    if sym = oddsym then
    begin getsym; expression(fsys); gen(opr,0,6)
    end else
    begin expression([eql,neq,lss,gtr,leq,geq]+fsys);
      if not (sym in [eql,neq,lss,leq,gtr,geq]) then
        error(20) else
        begin relop := sym; getsym; expression(fsys);
          case relop of
            eql: gen(opr,0, 8);
            neq: gen(opr,0, 9);
            lss: gen(opr,0,10);
            geq: gen(opr,0,11);
            gtr: gen(opr,0,12);
            leq: gen(opr,0,13);
          end
        end
    end
  end (*condition*) ;
```

```
begin (*statement*)
  if not (sym in fsys+[ident]) then
  begin error(10);
     repeat getsym until sym in fsys
  end ;
  if sym = ident then
  begin i := position(id);
     if i = 0 then error(11) else
     if table[i].kind <> variable then
        begin (*assignment to non-variable*) error(12); i := 0
        end ;
     getsym; if sym = becomes then getsym else error(13);
     expression(fsys);
     if i <> 0 then
        with table[i] do  gen(sto,lev-level,adr)
  end else
  if sym = callsym then
  begin getsym;
     if sym <> ident then error(14) else
        begin i := position(id);
           if i = 0 then error(11) else
           with table[i] do
              if kind = prozedure then gen(cal, lev-level, adr)
              else error(15);
           getsym
        end
  end else
  if sym = ifsym then
  begin getsym; condition([thensym,dosym]+fsys);
     if sym = thensym then getsym else error(16);
     cx1 := cx; gen(jpc,0,0);
     statement(fsys);  code[cx1].a := cx
  end else
  if sym = beginsym then
  begin getsym; statement([semicolon,endsym]+fsys);
     while sym in [semicolon]+statbegsys do
     begin
        if sym = semicolon then getsym else error(10);
        statement([semicolon,endsym]+fsys)
     end ;
     if sym = endsym then getsym else error(17)
  end else
  if sym = whilesym then
  begin cx1 := cx; getsym; condition([dosym]+fsys);
     cx2 := cx; gen(jpc,0,0);
     if sym = dosym then getsym else error(18);
     statement(fsys); gen(jmp,0,cx1); code[cx2].a := cx
  end ;
  test(fsys,[],19)
end (*statement*) ;
```

```
begin (*block*) dx := 3; tx0 := tx; table[tx].adr := cx; gen(jmp,0,0);
  if lev > levmax then error(32);
  repeat
    if sym = constsym then
    begin getsym;
      repeat constdeclaration;
        while sym = comma do
          begin getsym; constdeclaration
          end ;
        if sym = semicolon then getsym else error(5)
      until sym <> ident
    end ;
    if sym = varsym then
    begin getsym;
      repeat vardeclaration;
        while sym = comma do
          begin getsym; vardeclaration
          end ;
        if sym = semicolon then getsym else error(5)
      until sym <> ident;
    end ;
    while sym = procsym do
    begin getsym;
      if sym = ident then
        begin enter(prozedure); getsym
        end
      else error(4);
      if sym = semicolon then getsym else error(5);
      block(lev+1,tx,[semicolon]+fsys);
      if sym = semicolon then
        begin getsym; test(statbegsys+[ident,procsym],fsys,6)
        end
      else error(5)
    end ;
    test(statbegsys+[ident],declbegsys,7)
  until not (sym in declbegsys);
  code[table[tx0].adr].a := cx;
  with table[tx0] do
    begin adr := cx;  (*start adr of code*)
          size := dx; (*size of data segment*)
    end ;
  cx0 := cx; gen(int,0,dx);
  statement([semicolon,endsym]+fsys);
  gen(opr,0,0);  (*return*)
  test(fsys,[],8);
  listcode;
end (*block*) ;
```

```
procedure interpret;
   const stacksize = 500;
   var p,b,t: integer;  (*program-, base-, topstack-registers*)
      i: instruction;   (*instruction register*)
      s: array [1..stacksize] of integer;   (*datastore*)

   function base(l: integer): integer;
      var b1: integer;
   begin b1 := b;    (*find base l levels down*)
      while l > 0 do
         begin b1 := s[b1]; l := l-1
         end ;
      base := b1
   end (*base*) ;

begin writeln(" Start PL/0");
   t := 0; b := 1; p := 0;
   s[1] := 0; s[2] := 0; s[3] := 0;
   repeat i := code[p]; p := p+1;
      with i do
      case f of
    lit:  begin t := t+1; s[t] := a
          end ;
    opr: case a of       (*operator*)
          0: begin (*return*)
                t := b-1; p := s[t+3]; b := s[t+2];
             end ;
          1: s[t] := -s[t];
          2: begin t := t-1; s[t] := s[t] + s[t+1]
             end ;
          3: begin t := t-1; s[t] := s[t] - s[t+1]
             end ;
          4: begin t := t-1; s[t] := s[t] * s[t+1]
             end ;
          5: begin t := t-1; s[t] := s[t] div s[t+1]
             end ;
          6: s[t] := ord(odd(s[t]));
          8: begin t := t-1; s[t] := ord(s[t] = s[t+1])
             end ;
          9: begin t := t-1; s[t] := ord(s[t] <> s[t+1])
             end ;
         10: begin t := t-1; s[t] := ord(s[t] < s[t+1])
             end ;
         11: begin t := t-1; s[t] := ord(s[t] >= s[t+1])
             end ;
         12: begin t := t-1; s[t] := ord(s[t] > s[t+1])
             end ;
         13: begin t := t-1; s[t] := ord(s[t] <= s[t+1])
             end ;
         end ;
    lod: begin t := t+1; s[t] := s[base(l)+a]
```

```
        end ;
    sto: begin s[base(l)+a] := s[t]; writeln(s[t]); t := t-1
        end ;
    cal: begin (*generate new block mark*)
            s[t+1] := base(l); s[t+2] := b; s[t+3] := p;
            b := t+1; p := a;
        end ;
    int: t := t+a;
    jmp: p := a;
    jpc: begin if s[t] = 0 then p := a;  t := t-1;
        end
      end (*with, case*) ;
  until p = 0;
  write(" End PL/0");
end (*interpret*) ;


begin (*main program, for initializations see Program 4*)
   mnemonic[lit] := " LIT "; mnemonic[opr] := " OPR ";
   mnemonic[lod] := " LOD "; mnemonic[sto] := " STO ";
   mnemonic[cal] := " CAL "; mnemonic[int] := " INT ";
   mnemonic[jmp] := " JMP "; mnemonic[jpc] := " JPC ";
   declbegsys := [constsym,varsym,procsym];
   statbegsys := [beginsym,callsym,ifsym,whilesym];
   facbegsys := [ident,number,lparen];
   page(output); err := 0;
   cc := 0; cx := 0; ll := 0; ch := " "; kk := al; getsym;
   block(0,0,[period]+declbegsys+statbegsys);
   if sym <> period then error(9);
   if err = 0 then interpret else write(" Errors in PL/0 program");
99: writeln
end .
```

Programm 6: PL/0 Compiler

## 12. Konzeption und Implementation einer Spracherweiterung: Co-Prozesse

Programme ganz allgemein sind Beschreibungen von Automaten, die sich strikte nach einer vorgeschriebenen Regel verhalten. Wir betrachten den Computer als Werkzeug, das sich je nach Programm in einen anderen Automaten verwandeln lässt. Im Lichte dieser Betrachtungsweise erscheint das Programmieren als ein Konstruieren von Maschinen aus Bauteilen, die die verwendete Programmiersprache zur Verfügung stellt. Nun ist es in keinem anderen Zweig des Ingenieurwesens derart leicht, einmal konstruierte Maschinen abzuändern, zu erweitern, veränderten Bedürfnissen anzupassen. Dies hat zur Folge, dass sich der Programmierer wie kaum ein anderer Ingenieur auf die leichte, scheinbar kostenlose Wandelbarkeit seiner Produkte verlässt. So gibt es in der Praxis denn auch eher selten programmierte Systeme, die je den Zustand der Endgültigkeit erreichen. Besonders bei der Konstruktion komplexer Systeme ist es daher wichtig, sie so zu planen, dass ein Ausbau in verschiedenster Art möglich ist. Ist eine Grundstruktur glücklich gewählt, so lassen sich Aenderungen auf wenige Orte lokalisieren, was für die Uebersichtlichkeit und Zuverlässigkeit enorm wichtig ist.

Der Charakter der ständigen Evolution ist aber nicht nur Programmen eigen, sondern auch den Programmiersprachen. Dies erklärt sich leicht aus dem Umstand, dass deren Uebersetzer auch wieder wandelbare Programme sind. Dieses Kapitel sei daher einer solchen Spracherweiterung und deren Realisierung gewidmet. Sie ist aber lediglich *ein* Beispiel unter vielen für das viel allgemeinere Phänomen eines Systemausbaus. Die vorerst formulierten Regeln des Vorgehens haben denn auch Gültigkeit über das nachher verwendete Beispiel hinaus. Weitere, ähnlich Projekte finden sich unter den Aufgaben am Ende dieses Buches.

Mit grossem Vorteil wird bei einer beabsichtigten Erweiterung einer Programmiersprache die Aufmerksamkeit zuerst den konzeptionellen Aspekten gewidmet. Die neuen Sprachelemente sollen in einer allgemeingültigen Weise erfasst und definiert werden. Eine Ausrichtung auf eine bestimmte Art ihrer Mechanisierung ist wenn möglich zu vermeiden. Selbstverständlich kennt der erfahrene Konstrukteur gewisse Techniken der Implementation, und er weiss sogar deren Vor- und Nachteile einigermassen abzuschätzen. Die Kunst liegt aber vor allem darin, sich nicht allzu früh auf eine bestimmte Lösung festzulegen oder auszurichten. Wir stellen daher einen Plan auf, der angeben soll, worauf sich unsere Hauptaufmerksamkeit in zeitlicher Folge richten soll.

1. Definition der Grundkonzepte, die neu eingeführt werden sollen.
2. Definition einer Darstellung der neuen Sprachelemente. Formale Syntax und Semantik.
3. Festlegung der für die neuen Sprachelemente zu erzeugenden Befehlsfolgen. Falls nötig, Definition der Erweiterungen der interpretiereden Maschine.

4. Bestimmung der Datentypen und Datenstrukturen im Compiler, die zur Darstellung der neuen Sprachelemente notwendig sind.
5. Programmierung der Algorithmen, welche die neuen Sprachelemente verarbeiten. Integration der neuen Compilerteile.
6. Prüfung der Korrektheit der Implementation anhand von Testprogrammen mit und ohne neue Sprachelemente.

Wir durchlaufen nun diese Schritte mit Ausnahme der zwei letzten für einen beispielhaften Fall. PL/0 soll von einer rein sequentiellen in eine Sprache erweitert werden, in der gleichzeitige Programmabläufe beschrieben werden können.

Schritt 1: Grundkonzepte.
Wir weichen vom Axiom des rein sequentiellen Ablaufs von Programmen ab, indem wir die Vereinbarung von mehreren, in sich selber sequentiellen *Prozessen* zulassen, die, einmal gestartet, gleichzeitig ablaufen dürfen. Dieses Konzept ist aber nur dann von Interesse, wenn die Prozesse irgendwie miteinander kommunizieren können. Die einfachste Art dazu ergibt sich aus der Verwendung gemeinsamer Variablen. Damit lassen sich Prozesse auch synchronisieren, d.h. so programmieren, dass sie an bestimmten Punkten aufeinander warten. Nehmen wir z.B. an, dass ein Prozess P nach der Anweisung A auf die Beendigung der Anweisung B durch einen Prozess Q warten soll, so kann dies programmiert werden, indem eine gemeinsame Variable w wie folgt verwendet wird. w habe anfänglich den Wert *true*.

```
P:   ...   A;  while w do ;      ...
Q:   ...   B;  w := false;       ...
```

Auch wenn wir zulassen, dass solche Synchronisationen selten ausgeführt werden - wir sprechen in diesem Fall von *lose gekoppelten* Prozessen - so hat doch diese Technik der Synchronisierung einen entscheidenden Nachteil. Sie legt es nahe, dass ein Programm mit n logischen Prozessen durch n wirkliche Prozessoren ausgeführt werde. Weshalb? Eine Konzentration der gesamten Arbeit auf wenige oder sogar einen einzigen Prozessor ist unrealistisch, weil die Prozesse sogar "beschäftigt" bleiben, wenn sie nichts weiteres tun als warten müssen. Die Anweisung "while w do nichts" verkörpert eine perverse Art des ungeduldigen Wartens und verschleiert die wirkliche Absicht des Programmierers. Sie ist gewählt, weil kein zutreffendes Sprachelement vorhanden ist. Wir korrigieren diesen Missstand, indem wir eine neue Anweisung für das Warten einführen.

Dies bedingt hinwiederum die Einführung einer neuen Art von Objekten, nämlich von Zielen, auf die gewartet wird. Je nach Betrachtungsweise werden diese Ziele *Bedingungen* (conditions), auf deren Erfüllung gewartet wird, oder *Ereignisse* (events), auf deren Eintreffen gewartet wird, genannt. Weniger sinnvoll ist dafür die in der Literatur ebenfalls verbreitete Bezeichnung *Warteschlange* (queue), da sie auf eine mögliche Implementation hindeutet. Wir wählen hier die Bezeichnung *Signal*: ein Prozess wartet auf das Eintreffen eines Signals, das von einem anderen Prozess

ausgesendet wird. Damit ist auch angedeutet, dass eine zweite Operation mit Signalen benötigt wird, nämlich das Absenden von Signalen.

Die Synchronisation von Prozessen mit diesen neuen Objekten anstatt mit konventionellen Variablen hat den grossen Vorteil, dass Prozesse einander nicht beim Namen kennen müssen. Sie warten auf ein Signal, das unabhängig von der Identität des Absenders einen gewissen, zur Fortsetzung notwendigen Zustand des Systems ankündigt. Dieser Umstand ist bei Simulationen von Systemen mit diskreten Ereignissen besonders vorteilhaft. Die hier geplante Erweiterung erschliesst damit ein vielseitiges neues Anwendungsgebiet.

Entscheidend ist nun aber der Umstand, dass es durch das explizite Konzept der Signale möglich wird, ein System mit mehreren Prozessen durch einen Computer mit einem einzigen Prozessor zu implementieren. Die Gleichzeitigkeit wird dabei allerdings nur simuliert. Falls jedoch ein Prozess in den Wartezustand gelangt, so kann der Prozessor unverzüglich zur Ausführung anderer Prozesse eingesetzt werden. Es ist nicht nötig, stets zu prüfen, ob das Warten abgebrochen werden kann, denn die einzige Art, wie dies zustande kommen kann, ist durch das Absenden des betreffenden Signals durch den ausgeführten Prozess. Wartende Prozesse können daher unbeaufsichtigt gelassen werden und beanspruchen keine Prozessorzeit.

Grundsäztlich wäre es nötig, neben den Prozessen und Signalen noch ein weiteres Konzept neu einzuführen, um simultane Prozesse nützlich einsetzen zukönnen. Es ist dies die *gegenseitige Ausschliessbarkeit* von Prozessen aus bestimmten Abschnitten von Programmen. Aus Gründen, die später ersichtlich werden, soll hier jedoch nicht darauf eingegangen werden.

Gleichsam zum Ausgleich postulieren wir jedoch ein weiteres neues Sprachelement, welches zwar nicht notwendig, aber nützlich ist, besonders im Zusammenhang mit Anwendungen im Gebiet der Simulation. Es ist dies das Konzept der *Zeit*, dargestellt durch eine simulierte Systemzeit. Sie wird manifest durch eine neue Anweisung, die es gestattet, einen Prozess während einer bestimmten Dauer zu suspendieren.

Schritt 2: Formale Definition von Syntax und Semantik.
Wir beschränken uns darauf, die erweiterte Syntax als EBNF Produktionen anzugeben. Die Aenderungen betreffen nur die Symbole *block* und *statement*.

```
block =      ["const" ident "=" number {"," ident "=" number} ";"]
             ["var" ident {"," ident} ";"]
             ["signal" ident {"," ident} ";"]
             {("procedure" | "process") ident ";" block ":"}  statement.

statement =  [ "wait" ident | "send" ident | "hold" expression |
             ident ":=" expression | "call" ident |  . . .  ].
```

Prozess-Vereinbarungen besitzen dieselbe Form wie Prozedur-Vereinbarungen. Falls in einem Aufruf (call) ein Prozess-Bezeichner verwendet wird, so bedeutet diese

Anweisung die Erzeugung eines neuen Prozesses, der simultan (oder quasi-simultan) mit dem aufrufenden Prozess ausgeführt werden soll. Dies steht im Gegensatz zum Aufruf einer Prozedur, wo der aufrufende Prozess erst wieder aufgenommen wird, wenn die Prozedur beendet ist. Die *wait* und *send* Anweisungen beziehen sich auf ein Signal. Die Bedeutung einer Signalübermittlung liegt darin, dass dem Empfänger mitgeteilt wird, dass das System (von Prozessen) einen bestimmten Zustand erreicht hat. Diesen eigentlichen Informationsgehalt eines Signals s wollen wir durch das Prädikat $p_s$ bezeichnen. Die Semantik der beiden neuen Anweisungen kann jetzt axiomatisch definiert werden mittels einer Vorbedingung für *send* und einer Konsequenz für *wait*.

$$\{p_s\} \quad \text{send } s$$
$$\text{wait } s \quad \{p_s\}$$

Die Anweisung "**hold** n" postuliert eine simulierte Systemzeit, dargestellt durch eine ganzzahlige Grösse, die monoton zunimmt. Obige Anweisung verzögert den ausführenden Prozess um n Zeiteinheiten. In Simulationsprogrammen folgt sie üblicherweise einer Anweisung A, wenn besagt werden soll, dass in Wirklichkeit A eine bestimmte Zeit dauern soll, dass jedoch A als Einheit betrachtet wird, deren interner zeitlicher Ablauf keine Rolle spielt.

**Schritt 3:** Erweiterung der PL/0 Maschine und Definition des erzeugten Codes.
Es ist offensichtlich, dass die wesentliche konzeptionelle Erweiterung von PL/0 durch simultane Prozesse auch einen entsprechenden Ausbau der PL/0 Maschine erfordert. Wir gehen im Folgenden ausdrücklich von der Annahme aus, dass die Gleichzeitigkeit nicht nur simuliert werden kann, sondern sogar simuliert werden muss. Die PL/0 Maschine bleibt daher weiterhin als sequentielles Programm beschreibbar. Wir wenden uns aber vorerst nicht der Frage zu, wie ein einziger Prozessor mehrere Prozesse bedienen soll, sondern erwähnen zuerst die bei der Erzeugung eines Prozesses auftretenden Probleme. Sofort stellt sich dabei die Problematik der Speicherverwaltung in den Vordergrund. Wenn mehrere Prozesse gleichzeitig existieren und keine einfache Beziehung zwischen den zeitlichen Folgen ihrer Entstehung und Auflösung besteht, so kann die bewährte Stapelstrategie nicht aufrecht erhalten werden. Wir wollen allerdings hier nicht auf allgemeine Methoden der Speicherverwaltung eingehen und postulieren daher einige vereinfachende Regeln ohne jeglichen Anspruch auf Verwendbarkeit in praktischen Systemen.

1. Prozesse werden nur im Hauptprogramm vereinbart. Dadurch wird vermieden, dass ein Prozess Q Bezug auf lokale Variablen eines Prozesses P nehmen kann, in dem Q selbst lokal ist, nachdem sich P bereits aufgelöst hat.
2. Bei jeder Erzeugung eines neuen Prozesses wird ihm ein Speicherplatz fester Grösse zugeordnet. Darin befindet sich sein eigener Stapel.
3. Am Ende eines Prozesses wird der Speicher wieder an die Verwaltung zurückgegeben. Ist diese Verwaltung geschickt organisiert, so kann er bei einer späteren Prozesserzeugung wieder verwendet werden.

Der Hauptnachteil dieser primitiven Strategie liegt darin, dass der Stapel in einem festen Speicherplatz untergebracht werden muss, obwohl die grösste Ausdehnung eines Stapels nicht im voraus bekannt ist. Ihr Vorteil liegt aber in ihrer Einfachheit; insbesondere kann der Speicher weiterhin als Array dargestellt werden, und die einzelnen Stapel sind zusammenhängende Abschnitte dieses Arrays. Wir teilen nun den Datenspeicher S in Abschnitte ein, von denen jeder einem Prozess gehört, und deren erste Elemente einen Prozess-Descriptor bilden. Den Abschnitt nennen wir ein *Prozess-Segment*. Jedes Prozess-Segment besteht aus einzelnen Prozedur-Segmenten und ist als Stapel verwaltet. Die einzelnen Prozess-Segmente sind nach Fig.9 durch Zeiger (Indices) verkettet. Die Kette aller aktiven Segmente bildet den sogenannten *Prozess-Ring*. Das neue Register CP (current process) bezeichnet denjenigen Prozess, der sich gegenwärtig in Ausführung befindet.

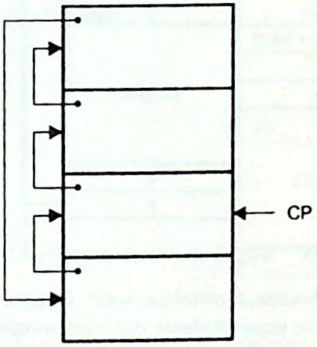



Fig. 9. Datenspeicher mit Prozess-Segmenten.

Zu beachten ist ferner, dass das Hauptprogramm selbst am einfachsten auch als Prozess betrachtet wird. Es ist vernünftig, ihm einen grösseren, der Anzahl globaler Variablen entsprechenden Speicherplatz zur Verfügung zu stellen.

Die Register P, T und B (siehe Kapitel 10) enthalten die entsprechenden Werte des gegenwärtig ausgeführten Prozesses. Die Registerwerte aller anderen Prozesse müssen im Segment abgespeichert werden. Das Schema eines Segmentes mit den entsprechenden Datenfeldern ist in Fig. 10 angegeben.

SL bezeichnet nach wie vor die statische Kette der Prozedur-Segmente (siehe Kapitel 10), *link* bezieht sich auf den Prozess-Ring, und *state* gibt den Zustand eines Prozesses an. Er ist entweder bereit (*state* = 0) oder wartet auf ein Signal (*state* > 0). Es liegt eigentlich nahe, dem wartenden Zustand als weiteres Attribut einen Verweis auf das erwartete Signal beizuordnen. Dies ist jedoch darum wenig sinnvoll, weil der wartende Prozess selbst gar nie in die Lage kommt, auf dieses Signal Bezug zu nehmen. Vielmehr ist es zweckmässig, umgekehrt dem Signal einen Verweis auf den Prozess zuzuordnen. Weil mehrere Prozesse auf ein und dasselbe Signal warten

können, ergibt sich im allgemeinen Fall eine Verkettung dieser Prozesse als geeignete Lösung. Der Anfang dieser Warteschlange von Prozessen bildet gleichsam den "Wert", der dem Signal zugeordnet ist. Jedem vereinbarten Signal wird daher vom Compiler wie einer Variablen ein Speicherplatz zugewiesen. Für den Programmierer hat das Signal jedoch keinen Wert, der gelesen und durch eine Zuweisung verändert werden könnte. Das Feld Q im Prozess-Descriptor dient zur Speicherung dieses Kettengliedes (queue). Der letzte Prozess in der Kette erhält den Zeigerwert Q = nil, dargestellt durch eine Null.

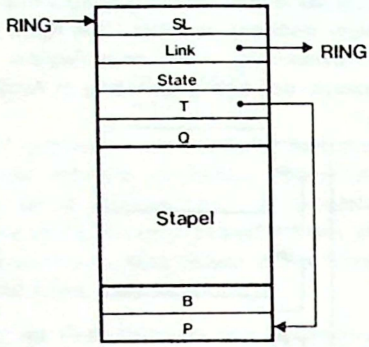



Fig. 10. Segment eines einzelnen Prozesses.

Nachdem nun die sogenannte Architektur oder Struktur der neuen PL/0 Maschine feststeht, müssen noch die neuen Befehle definiert werden. Es handelt sich um deren fünf, nämlich

1. den Befehl STRT, um einen neuen Prozess zu erzeugen und zu starten,
2. den Befehl SEND, um ein Signal abzusenden,
3. den Befehl WAIT, um auf ein Signal zu warten,
4. den Befehl HOLD, um einen Prozess zu verzögern, und
5. den Befehl STOP, um einen Prozess aufzulösen.

Der Befehl STRT besitzt analog zu CAL die Anfangsadresse eines Prozesses als Parameter. SEND und WAIT besitzen die Datenadresse des Signals als Parameter, während STOP und HOLD in der parameterlosen Klasse der Operatoren untergebracht werden können. Es ist zweckmässig, das Argument von HOLD als oberstes Element im Stapel zu belassen, da es als Resultat eines Ausdrucks auftreten kann.

Nachdem nun auch die Erweiterungen des Befehlsrepertoires definiert sind, bleiben noch die Algorithmen der einzelnen Befehle zu beschreiben. Wir verzichten auf eine ausführliche Programmierung und beschränken uns auf einige wesentlich Punkte. Die Implementation der Befehle STRT und STOP ist sehr einfach, wenn nach dem erwähnten Prinzip der Speicherzuteilung vorgegangen wird. Wir wenden uns daher sogleich den Befehlen SEND und WAIT zu, indem wir die Zustände eines

Prozess-Segmentes vor und nach der Befehlsausführung beschreiben.

Der Prozess wird in die Zeigerkette, die von s ausgeht und die Warteschlange der auf s wartenden Prozessen darstellt, eingereiht (s. Fig. 11). Nach der Suspendierung des Prozesses kann im Prinzip jeder beliebige Prozess, der den Zustand *bereit* aufweist, wieder aufgenommen werden. Da alle Prozesse im Ring verkettet sind, ist es naheliegend, entlang diesem Ring zu suchen und den ersten auszuwählen, der bereit ist. Existiert kein solcher Prozess, so befindet sich das System in einer Patt-Situation, wo jeder Prozess auf das Signal eines anderen wartet. Dieser Zustand wird als *deadlock* bezeichnet, und kommt in einem korrekt programmierten System nicht vor. Eine Fehlermeldung ist also in diesem Fall am Platz.

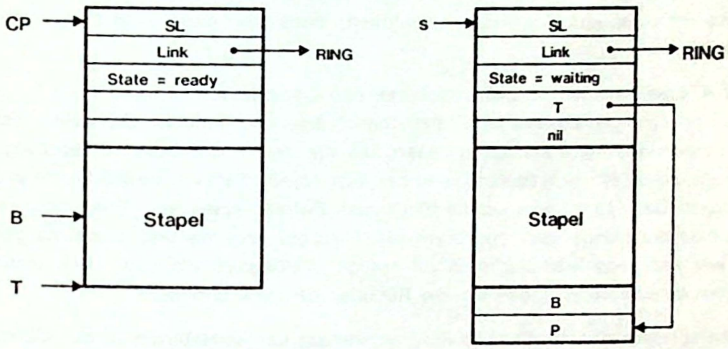



Fig. 11. Prozess-Segment vor und nach **wait** *s*

Bei der Implementation des Befehls SEND ist man geneigt, lediglich den ersten Prozess aus der Warteschlange herauszulösen und seinen Zustand als *bereit* zu markieren. Dies ist jedoch nicht zulässig, und zwar weil die für **wait** und **send** postulierten Axiome verlangen, dass der empfangende Prozess denselben System-Zustand vorfindet, den der Absender signalisiert. Würde nun der Prozessor nicht sogleich vom Absender an den wartenden Empfänger übergeben, so könnte ein dritter Prozess den signalisierten Zustand inzwischen wieder verändern. Daraus ergibt sich, dass der Befehl SEND ganz ähnlich wie WAIT implementiert werden muss, wobei allerdings der Sender den Zustand *bereit* beibehält und der Empfänger durch den Befehl explizite bezeichnet ist. Er muss also nicht im Ring zuerst noch gesucht werden.

Es bleibt zuletzt noch der Fall zu behandeln, wo die Warteschlange des gesendeten Signals leer ist. Wie ist die Wirkung eines Signals zu definieren, auf das kein Prozess wartet? Es ist üblich, sie in diesem Fall einfach als nichtig zu erklären und mit dem sendenden Prozess weiterzufahren. Allerdings gibt es auch Gründe, diesen Fall als unbeabsichtigten Programmierfehler zu betrachten und dementsprechend zu interpretieren.

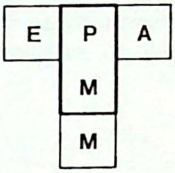
Der Befehl HOLD wurde eingeführt, um Anwendungen im Gebiet der Simulation diskreter Ereignisse zu ermöglichen. Es zeigt sich, dass die dazu nötigen Erweiterungen trivialer Natur sind. Der Schlüssel dazu liegt in der Einführung eines versteckten Signals; wir nennen es *tick* und weisen damit auf das Ticken einer Uhr hin. Die Operation **hold** wird sodann als "wait *tick*" interpretiert. Der zu verzögernde Prozess wird also in die Warteschlange der Prozesse eingereiht, die auf einen bestimmten Schlag der Uhr, d.h. einen bestimmten Zeitpunkt warten. In diesem Fall muss allerdings das Prozess-Segment angeben, welches dieser Zeitpunkt ist. Das Ticken der Uhr wird sodann ganz einfach realisiert, indem jedesmal, wo ein *deadlock* angezeigt wird, derjenige Prozess in der tick-Kette aufgeweckt wird, der den kleinsten Zeitwert anzeigt. Die Systemzeit wird auf diesen Wert fortgeschaltet, bevor der Prozess wieder aufgenommen wird. Es lohnt sich, bei der Ausführung von HOLD den Prozess derart in die tick-Kette einzufügen, dass sie stets nach Aufweckzeiten geordnet ist.

Schritt 4: Erweiterungen im Deklarationsteil des Compilers.
Die hier notwendigen Zusätze sind relativ beschränkt und einfach. Sicherlich wird der Wertebereich des Typs *symbol* erweitert, um die neuen Symbole zu repräsentieren (siehe *Programm 6*). Im weiteren erhält der Typ *object* die zwei neuen Werte *prozess* und *signal*. Der Typ *fct* wird um die fünf neuen Befehle erweitert. Erwähnenswert ist noch, dass die Menge *declbegsys* um die Symbole *prozess* und *signal* ausgedehnt wird, während *send*, *wait* und *hold* zur Menge *statbegsys* stossen. Neu unter den Variablen im Interpreter finden wir die Register CP, *tick* und *time*.

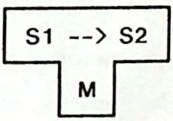
Wir überspringen eine ausführliche Programmierung der Erweiterungen des Compilers, die nicht aufwendig sind. In der Praxis wird genau diese Technik der Systemzeit und der Warteschlangen oft für Simulationssysteme verwendet. Allerdings gelangen meistens raffiniertere Strategien der Speicherverwaltung und des Einsatzes des Prozessors (process scheduling) zur Anwendung. Weil das Thema dieses Kapitels jedoch die Erweiterung von Sprachen und deren Implementation ist, soll auf Verbesserungsmöglichkeiten des partikulären Beispiels nicht eingegangen werden.

## 13. Technik der Compilerentwicklung und -Uebertragung

In den vorangehenden Kapiteln wurde ein PL/0 Compiler entwickelt und programmiert. Daraus wird ersichtlich, dass ein Compiler im allgemeinen ein komplexes Programm ist, und dass deshalb die Verwendung einer höheren Programmiersprache besonders zu empfehlen ist. Selbstverständlich gelten auch die Regeln und Praktiken der systematischen Programmentwicklung genau wie für jeden anderen komplizierten Algorithmus. Es gibt aber doch einige Punkte, die für Compiler spezifisch sind und deshalb hier noch erwähnt werden sollen. Insbesondere beziehen sich diese auf den relativ häufigen Fall der Erweiterung. Da ein Compiler, ja selbst eine Programmiersprache selten von Anfang an in endgültiger Fassung in Angriff genommen wird, wird vielmehr ein Kern der Sprache implementiert, worauf das Endziel schrittweise approximiert wird. Auf diese Technik soll nachfolgend näher eingegangen werden, doch führen wir vorerst noch eine Notation zur Darstellung von Programmentwicklungen ein. Das Bild




soll andeuten, dass Eingabedaten E mittels eines Programms P, formuliert in der Sprache M und daher interpretierbar durch einen Computer M, in Ausgabedaten A verwandelt werden. Handelt es sich beim Programm P um einen Compiler, der Sätze aus der Sprache S1 in die Sprache S2 übersetzt, so wird dieser Compiler, der selbst in der Sprache M formuliert ist, durch die Figur




dargestellt. Diese Form eines T hat denn auch die Bezeichnung *T-Diagramme* geprägt. Mit ihrer Hilfe können wir den üblichen Vorgang einer Compilation mit nachfolgender Ausführung eines Programmes P nach Fig. 12 schematisch beschreiben, wobei in beiden Schritten derselbe Computer M verwendet wird.

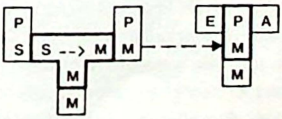



Fig.12. Schematische Darstellung von Compilation und Ausführung

In dieser Figur tritt nun die charakteristische Eigenschaft der T-Diagramme zutage: sie werden wie Dominosteine aneinander gereiht. Sich berührende Felder müssen dabei denselben Bezeichner tragen. Insbesondere wird deutlich, dass der Compiler das Programm P, formuliert in der Quellsprache S, in dasselbe Programm P, jedoch formuliert in der Zielsprache M, übersetzt. Als weiteres Beispiel einer Anwendung von T-Diagrammen zeigen wir den Vorgang einer Mehrphasen-Compilation in Fig. 13.

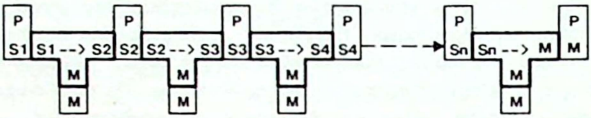



Fig. 13. n-Phasen Compilation eines Programms P

Der Vorgang einer interpretativen Ausführung eines Programms P, formuliert in einer Sprache (oder einem Code) H, durch einen am Computer M implementierten Interpreter, wird durch das Schema in Fig. 14 veranschaulicht. Seine Verallgemeinerung ist die mehrstufige Interpretation, ebenfalls dargestellt in Fig. 14. Ein Beispiel dafür wäre der PL/0 Code für H1, implementiert an einer mikroprogrammierten Maschine, deren Mikrocode H2 ist.

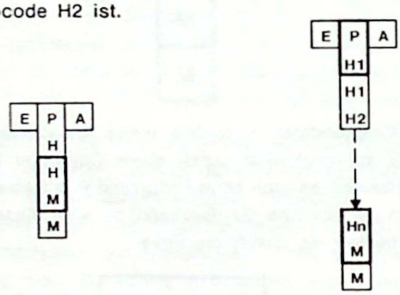



Fig. 14. Interpretative Ausführung eines Programms P

Nach dieser Einführung der T-Diagramme wenden wir uns nun der Technik der schrittweisen Compiler-Entwicklung zu. Hier ergeben sich nämlich beachtliche Vorteile, wenn ein *Compiler für eine Quellsprache S selbst auch in S formuliert* wird. Nehmen wir vorerst einmal an, dass ein Compiler für ein S existiere, und zwar sowohl in S selbst als auch in der Maschinensprache des zur Verfügung stehenden Computers M. Nun soll S in eine Version S' weiterentwickelt werden. Dies geschieht in zwei Schritten, nämlich (1) durch Programmierung der Erweiterungen in S (siehe Kapitel 12) und (2) durch Neuformulierung des Compilers, in der die neuen Sprachelemente nutzbringend verwendet werden. Der grosse Vorteil der Formulierung eines Compilers in seiner eigenen Quellsprache besteht also darin, dass der Compiler selbst von einer Erweiterung sofort profitieren kann. Deshalb kann die Entwicklung des Compilers mit einer relativ einfachen Sprache beginnen und durch

Wiederholung dieser zwei Schritte zum Endprodukt emporgehoben werden. Der Fachausdruck für dieses sich selbst Hochziehen lautet *"bootstrapping"*. Ein solcher Akt wird in Fig. 15 gezeigt, und lässt sich beliebig oft wiederholen.

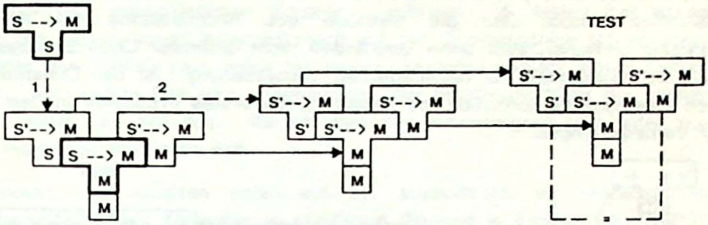



Fig. 15. Bootstrap eines Compilers von S nach S'

Erwähnenswert ist noch der dritte, in Fig. 15 abgebildete Schritt. Er dient zur Prüfung der Richtigkeit der Umprogrammierung des Compilers von S nach S'. Er erfolgt rein mechanisch, indem festgestellt wird, ob die Resultate der Schritte 2 und 3 genau identisch seien. Die Existenz eines solchen rigorosen Tests ist für die Praxis von unschätzbarem Wert.

Es stellt sich natürlich sofort die Frage nach dem Anfangsschritt. Wie wird ein Compiler für eine erste Version von S, formuliert in S, realisiert? Die übliche Lösung liegt in der Uebersetzung "von Hand" in eine andere, auf M verfügbare Programmiersprache H. Es hat sich als vorteilhaft erwiesen, dafür ebenfalls eine höhere Sprache anstelle von Assembler-Code zu verwenden. Der Anfangsschritt ist in Fig. 16 gezeigt.

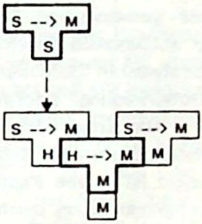



Fig. 16. Realisierung eines Compilers für S.

Der Anfangsschritt wird durch den glücklichen Umstand wesentlich erleichtert, dass die Programmierung in der "fremden" Sprache H durchaus ohne Rücksicht auf Effizienz erfolgen darf. Die Ineffizienz des Initialproduktes pflanzt sich nämlich nicht fort und macht sich nur im ersten Bootstrap bemerkbar, wonach das Hilfsprodukt weggeworfen werden kann.

Tatsächlich braucht ein Bootstrap nicht unbedingt eine Spracherweiterung zu beinhalten. Er kann sich auch auf eine *Verbesserung der Compilationsmethode* und

des erzeugten Code beziehen. Ein solcher Bootstrap wird in Fig. 17 veranschaulicht. In der Notation $S_{x \rightarrow y}$ M bezeichne x=a die Verwendung der alten, x=b der verbesserten Compilationsmethode oder Code-Generierung, y=a, dass der Compiler selbst durch die alte, y=b durch die neue Version des Compilers erzeugt wurde. Offensichtlich erlaubt also die Methode des Bootstrapping die anfängliche Verwendung einfacher, aber unter Umständen nicht optimaler Code-Erzeugung, und offeriert die Möglichkeit der nachträglichen Verbesserung. Ist der Compiler in der eigenen Quellsprache formuliert, so kommt er als erstes Programm in den Genuss dieser Verbesserungen.

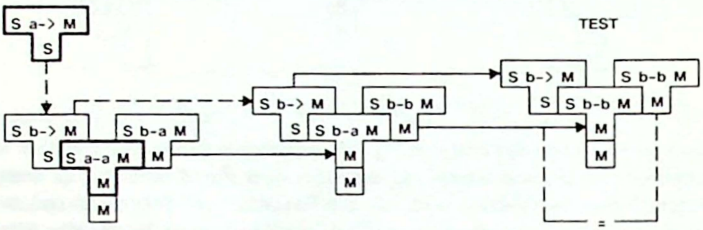



Fig. 17. Bootstrap zur Verbesserung eines Compilers.

Die angefügte Testphase erlaubt wiederum die Prüfung, ob die verbesserte Version des Compilers korrekt ist und sich selbst zu reproduzieren vermag.

Zum Abschluss sei noch auf den Problemkreis der Compiler-Uebertragung eingegangen. Eine Sprache S soll also auf einem Computer M2 verfügbar gemacht werden, indem ein existierender Compiler von M1 auf M2 übertragen wird. Compiler stellen unter allen Programmen gerade in dieser Beziehung einen Sonderfall dar. Die Verwendung einer M1 und M2 gemeinsamen Sprache bringt nämlich noch keine Lösung. Ein Compiler erzeugt schliesslich Befehlsfolgen seiner Zielsprache und ist damit naturgemäss auf einen bestimmten Zielcomputer zugeschnitten und somit nicht übertragbar (*portable*). Eine "Uebertragung" erfordert daher immer das Erstellen eines prinzipiell neuen Programmes. Allerdings können wesentliche Teile wie z.B. die Syntaxanalyse und die Fehlerbehandlung unverändert übernommen werden, falls der Compiler zweckmässig organisiert ist (siehe Kapitel 8 und 9). Dennoch ist dieser wesentliche Schritt einer Umprogrammierung unumgänglich.

Die naheliegende Lösung, die verbleibenden Arbeiten auf ein Minimum zu reduzieren, liegt natürlich in der Verwendung einer gemeinsamen Programmiersprache. Ein häufiger Kandidat für diese Rolle ist *Fortran*, doch zeigt sich in der Praxis immer wieder, dass Fortran Programme weit weniger transportabel sind als anfänglich angenommen wird. Dies rührt vor allem davon her, dass der Systemprogrammierer gezwungen ist, Erweiterungen zu benutzen, die nur für seine Anlage und deren Fortran Version gelten. Ferner bietet Fortran wenig Schutz vor der Nutzbarmachung bestimmter Eigenschaften der verwendeten Anlage, die durch Fortran hindurchschimmern. Als Folge davon ist die nachträgliche Anpassung eines

Programmes an eine andere Anlage oft ein aufwendiges Unterfangen.

Wir wollen daher zeigen, welche Techniken im Fall verwendet werden, wo wiederum der Compiler in seiner eigenen Sprache formuliert ist. Wir unterscheiden zwei Fälle, die grundsätzlich verschiedenes Vorgehen bedingen. Im ersten Fall werden die Arbeiten der Uebertragung ausschliesslich auf der Quellmaschine M1 vorgenommen. Dies dürfte dann üblich sein, wenn M1 einem Liferanten und M2 einem Auftraggeber gehören. Im zweiten Fall wird angenommen, der Empfänger führe die notwendigen Arbeiten selber auf M2 aus. Es ist dies der aufwendigere und unter normalen Verhältnissen kostspieligere Fall.

Angenommen, die Arbeiten seien auf M1 auszuführen, so benötigen wir als Ausgangsprodukte den Compiler in Quellform (S) und in binärer Form (M1). Das Vorgehen ist in Fig. 18 illustriert. Für die Testphase ist ein Interpreter (Emulator) von M2 auf M1 unerlässlich.

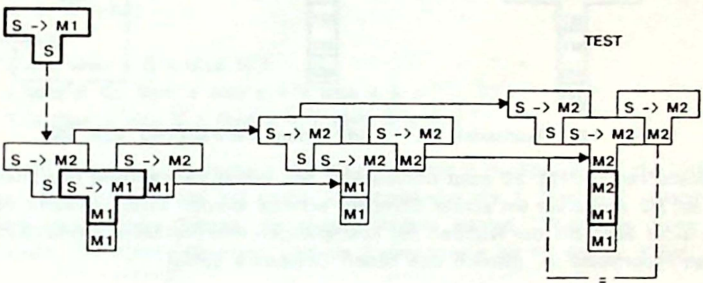



Fig. 18. Transport M1 → M2 unter Verwendung von M1.

Viel ausgeprägter wird von der Technik der Emulation Gebrauch gemacht, wenn die Arbeiten ausschliesslich auf dem Zielcomputer M2 durchgeführt werden müssen. Hier wird ein Interpreter (Emulator) von M1 auf M2 benötigt (siehe Fig. 19). Der Nachteil dieser Methode ist die ausschliessliche Verwendung interpretativer Compilationen. Immerhin ist zu bedenken, dass sogar ein guter Interpreter für jeden interpretierten Befehl im Durchschnitt zwischen 20 und 30 Maschinenbefehle ausführt. Berücksichtigt man ferner, dass in der Praxis beim Programmieren immer wieder Fehler unterlaufen, und dass daher jeder Schritt in Wirklichkeit etliche Male wiederholt wird, so lassen sich daraus Schlüsse auf den hohen Bedarf an Rechenleistung ziehen. Diese Methode ist daher für die Praxis nicht attraktiv, zumal die zunehmende Verfügbarkeit anderer Computertypen in Service-Zentren mittels Datenfernverbindungen bessere Lösungen offeriert.

Die interpretative Methode besitzt aber doch auch Qualitäten, wenn sie am richtigen Ort eingesetzt wird. So fällt die relativ langsame Ausführung von sehr kurzen Programmen wenig ins Gewicht, weil deren Compilations-, Lade-, und Ein/Ausgabezeiten die eigentliche Rechenzeit dominieren. Dies hat die Konsequenz, dass die Wahl eines hypothetischen Computers H und der Bau eines Compilers für S

auf H das Problem der Uebertragbarkeit wesentlich erleichtern kann. Die einzige Arbeit, die in diesem Fall vom Empfänger aufzuwenden ist, besteht in der Programmierung eines effizienten Interpreters für H auf der Zielmaschine. Der Compiler in Quellform (S) und in compilierter Form (H) bildet zusammen mit einer genauen Beschreibung der hypothetischen Maschine H gleichsam einen Baukasten (compiler kit). Die Uebertragung mit Hilfe eines solchen Baukastens stellt eine verhältnismässig einfache Angelegenheit dar.

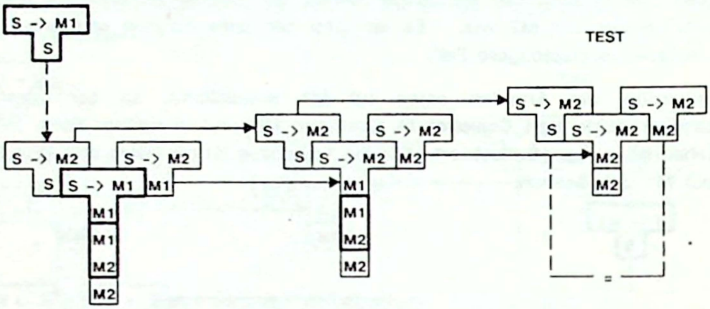



Fig. 19. Transport M1 → M2 unter Verwendung von M2.

Der untere Teil von Fig. 20 zeigt schliesslich, wie durch Verwendung des Interpreters für H auf M2 wiederum ein echter Compiler erzeugt werden kann. Diesem Vorgehen haftet leider ebenfalls der Nachteil der kostspieligen interpretativen Compilation eines grossen Programms an, nämlich des neuen Compilers selbst.

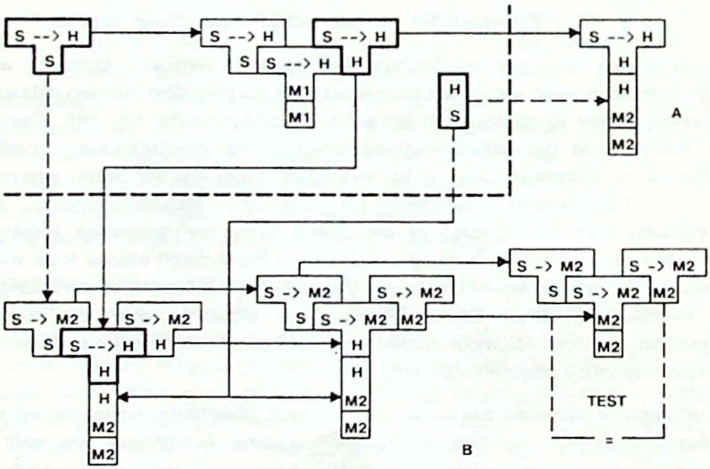



Fig. 20. Transport M1 → M2 mittels einer hypothetischen Maschine H.

## 14. Aufgabensammlung

1. Man untersuche die folgende Syntax:

```
S  =  A.
A  =  B | "if" A "then" A "else" A.
B  =  C | B "+" C | "+" C.
C  =  D | C "&" D | "&" D.
D  =  "x" | "(" A ")" | "%" D.
```

Bestimme die Mengen der terminalen und nicht-terminalen Symbole, sowie die Mengen der Anfangs- und Folgesymbole für jedes nicht-terminale Symbol. Konstruiere explizit die Folgen von Zerlegungsschritten für die Sätze

```
x + x
(x + x) & (+&x)
(x&%+x)
if x + x then x & x else % x
if x then if %x then x else x + x else x & x
if % x then x else if x then x + x else % x + x
```

2. Erfüllt die Syntax von Aufgabe 1 die Regeln 1 und 2 (s. Kap. 2) für die Zerlegung nach dem *top-down* Prinzip mit einem *lookahead* von nur 1 Symbol ? Wenn nicht, finde eine aequivalente Syntax, die diesen Regeln genügt. Stelle sie durch EBNF Produktionen, als Syntax-Graphen, und als Datenstruktur für *Programm 3* dar.

3. Zerlege das PL/0 Programm aus Kapitel 7 explizit in seine syntaktischen Komponenten, und zwar nach dem Prinzip des *top-down Parsing*.

4. Die folgenden Produktionen definieren einen Ausschnitt aus der Originalversion der Sprache *Algol 60*. Sie enthält einige Mehrdeutigkeiten, die später in der revidierten Fassung eliminiert wurden. Man ermittle diese Mehrdeutigkeiten anhand einiger Sätze und schlage eine alternative Syntax vor, welche der Sprache eine eindeutige Struktur zuweist.

```
expression     =  simpleexpression |
                  simpleexpression relation simpleexpression |
                  "if" expression "then" expression "else" expression .
relation       =  "=" | "≠" .
simpleexpression = "a" | "b" | . . . | "z" .

statement      =  basicstatement | ifstatement | forstatement.
ifstatement    =  "if" expression "then" statement |
                  "if" expression "then" statement "else" statement.
forstatement   =  "for" forlist "do" statement.
basicstatement =  "A" | "B" | . . . | "Z" .
```

forlist = "α" | "β" | . . . | "ω" .

Man untersuche die folgenden Sätze:

if a then b else c = d
if a then if b then A else B
if a then for α do if b then A else B

5. Algol 60 enthält eine Mehrfach-Zuweisung von der Form

v1 := v2 := . . . vn := e

Sie sei durch folgende Syntax definiert:

```
assignment  =  leftpartlist expression.
leftpartlist  =  leftpart | leftpartlist leftpart.
leftpart      =  variable ":=" .
expression  =  variable | expression "+" variable.
variable      =  ident | ident "[" expression "]" .
```

Welches ist der Grad des notwendigen "*lookahead*", um diese Syntax nach dem *top-down* Prinzip zu analysieren? Man schlage eine alternative Notation für die Mehrfach-Zuweisung vor, deren lookahead nur 1 Symbol ist.

6. Erstelle ein Programm, welches eine durch EBNF Produktionen dargestellte Syntax einliest und die Mengen der Anfangs-, Folge-, und Endsymbole ermittelt. Ein *Endsymbol* eines nichtterminalen Symbols A sei ein Symbol, das am Ende eines aus A erzeugten Satzes stehen kann.
*Hinweis*: Man benütze Warshall's Algorithmus (R.W.Floyd, Algorithm 96, Comm. ACM, June 1962).

```
type matrix = array [1..n, 1..n] of Boolean;

procedure ancestor(var m: matrix; n: integer);
(* Initially m[i,j] is true, if individual i is a parent of individual j.
  At completion, m[i,j] is true, if i is an ancestor of j *)
   var i,j,k: integer;
begin for i := 1 to n do
        for j := 1 to n do
            if m[j,i] then
                for k := 1 to n do
                    if m[i,k] then m[j,k] := true
end
```

7. Aendere *Programm 3* so, dass geprüft wird, ob die eingelesene Syntax den einschränkenden Regeln 1 und 2 genügt. *Programm 3* würde z.B. die Syntax

```
A  =  B C.
B  =  "x" "y" .
C  =  "x" "z" .
```

oder auch

A = {"x"} | {"y"}.

akzeptieren. Man untersuche zuerst, in welcher Art der Parser in diesen Fällen versagt.

8. Ermittle den PL/0 Code, den der Compiler (*Programm 6*) für die Prozeduren *divide* und *gcd* (Kap. 7) erzeugt.

9. Man verbessere das PL/0 System von *Programm 6*, so dass keine Programmierfehler zum Scheitern des Systems führen können. Man beachte insbesondere die Möglichkeiten eines zu kleinen Speichers und eines arithmetischen Ueberlaufs.

10. Ergänze PL/0 durch eine Ein- und Ausgabeanweisung. Programmiere die notwendigen Erweiterungen von *Programm 6*.

```
statement  =  "read" variable {"," variable} |
              "write" expression {"," expression} | . . .
variable   = ident.
```

11. Verändere die Sprache PL/0 in eine Variante PL/0', indem die bedingte und die repetierte Anweisung neu formuliert werden:

```
statement = [ ident ":=" expression | "call" ident |
              "if" condition "then" sequence
              {"elsif" condition "then" sequence} ["else" sequence] "end" |
              "while" condition "do" sequence
              {"orwhile" condition "do" sequence} "end" ] .
sequence = statement {";" statement} .
```

Die neue Anweisung

while a do A orwhile b do B . . . orwhile z do Z end

soll besagen, dass von allen Bedingungen a, b ... z, die erfüllt sind, eine beliebige ausgewählt und die zugehörige Anweisung ausgeführt wird. Dieser Vorgang wiederholt sich, bis keine der Bedingungen mehr erfüllt ist. Hinweis: Die Syntax der Prozedurvereinbarung sollte den neuen Umständen wie folgt angepasst werden.

"procedure" ident ";" sequence "end" ";"

12. Erweitere Sprache und Compiler PL/0 mit dem Datentyp *Boolean* und den logischen Operatoren and, or, und not. Die Syntax ist derart anzupassen, dass anstelle von Bedingungen (conditions) Bool'sche Ausdrücke zugelassen sind. Relationen können als Bool'sche Operanden in Ausdrücken auftreten.

13. Erweitere Sprache und Compiler PL/0 um den Datentyp *real* mit den arithmetischen Operatoren +, -, *, und /. Wähle zwischen den folgenden Alternativen: (a) Das Resultat einer Operation ist stets vom gleichen Typ wie die

Operanden. Die Typen *integer* und *real* lassen sich nicht "mischen". Hingegen gibt es Transferoperatoren zwischen den beiden Typen. (b): Operanden der Typen *integer* und *real* lassen sich beliebig mischen. Bei Zuweisungen zu ganzzahligen Variablen wird automatisch gerundet.

14. Erweitere die Sprache PL/0 durch Arrays unter der Annahme, dass die Indexgrenzen in der Vereinbarung der Array-Variablen angegeben werden. Die Grenzen seien wie folgt als Konstanten m und n spezifiziert:

var a [m : n]

[0 : max]

15. Erweitere die Sprache PL/0 durch Prozeduren mit Parametern. Man untersuche die folgenden beiden Möglichkeiten und wähle eine für die Implementation aus.

1. Werteparameter (*call by value*): Die aktuellen Parameter im Aufruf sind Ausdrücke. Sie werden beim Aufruf ausgewertet. Die formalen Parameter stellen lokale Variablen der Prozedur dar, denen beim Aufruf die Werte der entsprechenden aktuellen Parameter zugewiesen werden.

2. Adressparameter (*call by reference*): Die aktuellen Parameter sind Variablen, die beim Aufruf für die formalen Parameter substituiert werden. Solche Parameter werden implementiert, indem die Adressen der aktuellen Variablen in Speicherzellen abgelagert werden, die den formellen Parametern zugewiesen sind. Die Variablen werden dann indirekt adressiert. Damit wird es möglich, Variablen ausserhalb der Prozedur mittels der Parameter zu ändern. Es ist damit sinnvoll, die Scope-Regeln der Sprache den neuen Umständen anzupassen: Innerhalb von Prozeduren können nur lokale Grössen direkt zugegriffen werden; nicht-lokale Variablen sind durch Parameter zu erreichen.

16. Verbessere das PL/0 System (*Programm* 6) durch Einführung eines sogenannten *Display*. Damit soll der Zugriff zu Variablen beschleunigt werden. Das Display ist ein Array D von Indices, dessen Komponenten D[i] den Ort (Index) des Prozedur-Segmentes auf Niveau i im Stapel angeben. Damit erübrigt sich das Durchlaufen der SL-Kette beim Variablenzugriff, da sich die Adresse aus der einfachen Formel D[i]+a ergibt, wenn i und a Niveau und Offset der Variablen sind. Hingegen muss nun beim Aufruf und am Ende jeder Prozedur das Display auf den neuesten Stand gebracht werden. Kann das Display die SL-Kette ersetzen, oder ist sie dennoch nötig?

17. Führe die in Kapitel 12 behandelte Erweiterung von PL/0 durch Prozesse aus. Um ein brauchbares Werkzeug der Simulation zu erhalten, ist es unumgänglich, gleichzeitig auch Prozessparameter und Arrays einzuführen.

18. Aendere das PL/0 System derart, dass anstelle eines Stapels im Interpreter eine konventionelle Computer-Architektur verwendet wird. Dies bedeute insbesondere, dass keine auf den Stapelmechanismus ausgerichtete Befehle mehr vorhanden sind. Es werde vielmehr angenommen, dass mehrere Register zur Verfügung stehen, und dass die Befehle neben einer Speicheradresse noch eine Registernummer enthalten,

die das zu verwendende Register angibt.

19. Eliminiere den Interpreter aus *Programm 6* und ändere den Compiler derart, dass Code für einen bestehenden Computer erzeugt wird. Es empfiehlt sich, der Einfachheit halber (mindestens in einem ersten Schritt) symbolischen Assembler-Code zu erzeugen.

20. Erweitere *Programm 5* derart, dass es ein beliebiges PL/0 Programm in ein entsprechendes PL/0' Programm übersetzt. Die Syntax von PL/0' ist der Aufgabe ~~10~~ 11 zu entnehmen. Ist eine einfache Uebersetzung in der umgekehrten Richtung ebenfalls möglich?

## Literaturhinweise

Bauer, F.L. et al: *Compiler Construction, An Advanced Course.* Lecture Notes in Computer Science, 21. Springer-Verlag, Berlin, Heidelberg, New York, 1974.

Brinch Hansen, P. *Operating Systems Principles.* Prentice-Hall, Englewood Cliffs, 1973.

Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

Dijkstra, E.W., Dahl, O.-J., and Hoare, C.A.R. *Structured Programming.* Academic Press, London and New York, 1974.

Earley, J. and Strurgis, H. A formalism for translator interactions. *Comm. ACM 13*, 10 (Oct. 1970), 607-617.

Gries, D. *Compiler Construction for Digital Computers.* McGraw-Hill, New York, 1971.

Jensen, K. and Wirth, N. *PASCAL - User Manual and Report.* Springer Study-Editions, New York, 1975.

Kaubisch, W.H., Perrott, R.H. and Hoare, C.A.R. Quasiparallel programming. *Software - Practice and Experience 6*, 3 (July 1976), 341-356.

Knuth, D.E. Top-down syntax analysis. *Acta Informatica 1* (1971), 79-110.

Rohlfing, H. *SIMULA - Eine Einführung.* BI Hochschultaschenbücher, Bd. 747, Bibliographischer Verlag, Mannheim 1975.

Wirth, N. *Algorithmen und Datenstrukturen.* Teubner-Verlag, Stuttgart, 1975.

Wulf, W., *et al. The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

# Anhang

## Der ASCII Zeichensatz

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | nul | dle | | 0 | @ | P | ' | p |
| 1 | soh | dc1 | ! | 1 | A | Q | a | q |
| 2 | stx | dc2 | " | 2 | B | R | b | r |
| 3 | etx | dc3 | # | 3 | C | S | c | s |
| 4 | eot | dc4 | $ | 4 | D | T | d | t |
| 5 | enq | nak | % | 5 | E | U | e | u |
| 6 | ack | syn | & | 6 | F | V | f | v |
| 7 | bel | etb | ' | 7 | G | W | g | w |
| 10 | bs | can | ( | 8 | H | X | h | x |
| 11 | ht | em | ) | 9 | I | Y | i | y |
| 12 | lf | sub | * | : | J | Z | j | z |
| 13 | vt | esc | + | ; | K | [ | k | { |
| 14 | ff | fs | , | < | L | \ | l | \| |
| 15 | cr | gs | - | = | M | ] | m | } |
| 16 | so | rs | . | > | N | ↑ | n | ~ |
| 17 | si | us | / | ? | O | _ | o | del |

## Zweierpotenzen

| | | |
|---|---|---|
| 2 | 01 | .5 |
| 4 | 02 | .25 |
| 8 | 03 | .125 |
| 16 | 04 | .0625 |
| 32 | 05 | .03125 |
| 64 | 06 | .015625 |
| 128 | 07 | .0078125 |
| 256 | 08 | .00390625 |
| 512 | 09 | .001953125 |
| 1024 | 10 | .0009765625 |
| 2048 | 11 | .00048828125 |
| 4096 | 12 | .000244140625 |
| 8192 | 13 | .0001220703125 |
| 16384 | 14 | .00006103515625 |
| 32768 | 15 | .000030517578125 |
| 65536 | 16 | .0000152587890625 |
| 131072 | 17 | .00000762939453125 |
| 262144 | 18 | .000003814697265625 |
| 524288 | 19 | .0000019073486328125 |
| 1048576 | 20 | .00000095367431640625 |
| 2097152 | 21 | .000000476837158203125 |
| 4194304 | 22 | .0000002384185791015625 |
| 8388608 | 23 | .00000011920928955078125 |
| 16777216 | 24 | .000000059604644775390625 |
| 33554432 | 25 | .0000000298023223876953125 |
| 67108864 | 26 | .00000001490116119384765625 |
| 134217728 | 27 | .000000007450580596923828125 |
| 268435456 | 28 | .0000000037252902984619140625 |
| 536870912 | 29 | .00000000186264514923095703125 |
| 1073741824 | 30 | .000000000931322574615478515625 |
| 2147483648 | 31 | .0000000004656612873077392578125 |
| 4294967296 | 32 | .00000000023283064365386962890625 |
| 8589934592 | 33 | .000000000116415321826934814453125 |
| 17179869184 | 34 | .0000000000582076609134674072265625 |
| 34359738368 | 35 | .00000000029103830456733703613281 25 |
| 68719476736 | 36 | .000000000014551915228366851806640625 |
| 137438953472 | 37 | .0000000000072759576141834259033203125 |
| 274877906944 | 38 | .0000000000036379788070917129516601 5625 |
| 549755813888 | 39 | .0000000000018189894035458564758300 78125 |
| 1099511627776 | 40 | .00000000000090949470177292823791503 90625 |
| 2199023255552 | 41 | .000000000000454747350886464118957519 53125 |
| 4398046511104 | 42 | .0000000000002273736754432320594787 59765625 |
| 8796093022208 | 43 | .00000000000011368683772161602973937 98828125 |
| 17592186044416 | 44 | .000000000000056843418860808014869689 94140625 |
| 35184372088832 | 45 | .0000000000000284217094304040074348 44970703125 |
| 70368744177664 | 46 | .00000000000001421085471520200371742 24853515625 |
| 140737488355328 | 47 | .000000000000007105427357601001858711 24267578125 |
| 281474976710656 | 48 | .0000000000000035527136788005009293 5562133789 0625 |

## Sachverzeichnis


ASCII 91
Basisadress-Register 58
backtracking 10
Bedeutungsstruktur 14
Befehlssatz 59
Bezeichner-Tabelle 40
Bezugsdiagramm 41
BNF 6
bootstrap 81
bottom-up 11
call by value 88
call by reference 88
*Chomsky, N.* 8
deadlock 77
deterministisch 17
EBNF 26
Emulator 83
erweiterbare Sprache 33
first($\sigma$) 12
fixup 61
follow($\sigma$) 13
gegenseitiger Ausschluss 73
infix 60
Interpreter 57, 80
kontext-abhängig 8, 40
kontext-frei 8
Leitknoten 24
lookahead 10
Mehrphasen-Compiler 61, 80
Metasymbol 7
nicht-terminales Symbol 6
offset 58
Parser (Satzzerleger) 10, 24
postfix 60
Produktion 6
Prozedur-Segment 58, 75
Prozess-Ring 75
Prozess-Segment 75

Rekursion 8, 41
Satz 6
Scanner 27
Schlüsselwort 49
Segment-Descriptor 58
Semantik 6
Sprache 7
Signal 72
Stapel 40, 57
Stopsymbol 49
Symbol 6
Synchronisation 72
Syntax 6
T-Diagramm 79
Terminalsymbol 6
top-down 10
Vokabular 6
Warteschlange 72
Zeichen 27

## Korrekturen zu Wirth, Compilerbau

| Seite | Zeile | alter Text | neuer Text |
|---|---|---|---|
| 6 | -6 | hergeletet | hergeleitet |
| 7 | 3 | Strukur | Struktur |
| 13 | 18 | welchen | welchem |
| 16 | 6 und 12 | nicht terminal | terminal |
| 17 | 3 | alternative | Alternative |
| 20 | 2 und 20 | 4 | 5 |
| 24 | -11 | Koten einem | Knoten ein |
| 27 | 16 | verfizieren | verifizieren |
| 31 | 12 | p,q: | p,q,r,s: |
| 31 | -7 | q↑.alt := c | q↑.alt := a |
| 33 | -9 | wie er dem dem | wie er dem |
| 34 | -3 | Kapitels | Buches |
| 35 | 8 | komlizierter | komplizierter |
| 39 | 6 | genügen | genüge |
| 39 | 15 | do | do = ... |
| | 16 | do | do = + ... |
| | 17 | do | do = + * ... |
| 48 | -7 | entscheindend | entscheidend |
| 51 | 10-16 | (auslassen) | |
| 57 | 7 | Code Generierung | Code-Generierung |
| 62 | 13 | OPR 0,7 | OPR 0,6 |
| 71 | -17 | ähnlich | ähnliche |
| | -1 | interpretiereden | interpretierenden |
| 73 | 20 | zukönnen | zu können |
| 74 | 22 | ausdrüclkich | ausdrücklich |
| 78 | 13 | der tick-Kette | die tick-Kette |
| 83 | 6 | Liferanten | Lieferanten |
| 89 | 7 | 10 | 11 |
| 93 | 18 | 63 | 83 |
| | 22 | 12 | 13 |

B. G. Teubner Stuttgart

# Teubner Studienbücher Fortsetzung

K 16.86

## Mathematik Fortsetzung

Fischer/Sacher: **Einführung in die Algebra**
238 Seiten. DM 16,80

Grigorieff: **Numerik gewöhnlicher Differentialgleichungen**
Band 1: Einschrittverfahren. 202 Seiten. 14,80 DM
Band 2: Mehrschrittverfahren

Hainzl: **Mathematik für Naturwissenschaftler**
311 Seiten. DM 29,– (LAMM)

Hilbert: **Grundlagen der Geometrie**
11. Aufl. VII, 271 Seiten. DM 19,80

Jaeger/Wenke: **Lineare Wirtschaftsalgebra**
Eine Einführung
Band 1: XVI, 174 Seiten. DM 18,80 (LAMM)
Band 2: IV, 160 Seiten. DM 18,80 (LAMM)

Kall: **Mathematische Methoden des Operations Research**
Eine Einführung. 176 Seiten. DM 22,80 (LAMM)

Kochendörffer: **Determinanten und Matrizen**
IV, 148 Seiten. DM 16,80

Krabs: **Optimierung und Approximation**
208 Seiten. DM 24,80

Stiefel: **Einführung in die numerische Mathematik**
5. Aufl. 292 Seiten. DM 24,80 (LAMM)

Stummel/Hainer: **Praktische Mathematik**
299 Seiten. DM 28,80

Topsøe: **Informationstheorie**
Eine Einführung. 88 Seiten. DM 12,80

Velte: **Direkte Methoden der Variationsrechnung**
Eine Einführung unter Berücksichtigung von Randwertaufgaben bei partiellen Differentialgleichungen. 198 Seiten. DM 24,80 (LAMM)

Walter: **Biomathematik für Mediziner**
148 Seiten. DM 14,80

Witting: **Mathematische Statistik**
Eine Einführung in Theorie und Methoden. 2. Aufl. 223 Seiten. DM 26,80 (LAMM)

Preisänderungen vorbehalten

# Teubner Studienbücher Fortsetzung

K 16 80

## Mathematik Fortsetzung

Fischer/Sacher: **Einführung in die Algebra**
238 Seiten. DM 16,80

Grigorieff: **Numerik gewöhnlicher Differentialgleichungen**
Band 1: Einschrittverfahren. 202 Seiten. 14,80 DM
Band 2: Mehrschrittverfahren

Hainzl: **Mathematik für Naturwissenschaftler**
311 Seiten. DM 29,– (LAMM)

Hilbert: **Grundlagen der Geometrie**
11. Aufl. VII, 271 Seiten. DM 19,80

Jaeger/Wenke: **Lineare Wirtschaftsalgebra**
Eine Einführung
Band 1: XVI, 174 Seiten. DM 18,80 (LAMM)
Band 2: IV, 160 Seiten. DM 18,80 (LAMM)

Kall: **Mathematische Methoden des Operations Research**
Eine Einführung. 176 Seiten. DM 22,80 (LAMM)

Kochendörffer: **Determinanten und Matrizen**
IV, 148 Seiten. DM 16,80

Krabs: **Optimierung und Approximation**
208 Seiten. DM 24,80

Stiefel: **Einführung in die numerische Mathematik**
5. Aufl. 292 Seiten. DM 24,80 (LAMM)

Stummel/Hainer: **Praktische Mathematik**
299 Seiten. DM 28,80

Topsoe: **Informationstheorie**
Eine Einführung. 88 Seiten. DM 12,80

Velte: **Direkte Methoden der Variationsrechnung**
Eine Einführung unter Berücksichtigung von Randwertaufgaben bei partiellen Differentialgleichungen. 198 Seiten. DM 24,80 (LAMM)

Walter: **Biomathematik für Mediziner**
148 Seiten. DM 14,80

Witting: **Mathematische Statistik**
Eine Einführung in Theorie und Methoden. 2. Aufl. 223 Seiten. DM 26,80 (LAMM)

Preisänderungen vorbehalten

Leitfäden der angewandten Mathematik und Mechanik LAMM

Eine Einführung in die Grundlagen der formalen Sprachen und die Technik des Compilerbaus für Informatiker, Mathematiker und Ingenieure

Syntax und Satzzerlegung

Tabellengesteuerte Syntaxanalyse

Syntaxgraphen

Die Programmiersprache PL/O

Bau eines PL/O Compilers

Systematische Behandlung von syntaktischen Fehlern

Methodik der Erweiterung von Sprachen und Compilern

Technik der Compilerentwicklung und -übertragung

B.G.Teubner Stuttgart

ISBN 3-519-02338-5